Paulo Octávio Ribeiro Monteiro

licenciado em Engenharia Electrotécnica

pela

FACULDADE DE ENGENHARIA

UNIVERSIDADE DO PORTO

Concepção e Implementação de um Sistema Simulador de Ambientes Industriais

681.3(043)/MONP/CON

UNIVERSIDADE DO PORTO
Faculdade de Engenharia
BIBLIOTECA
N.º 27073
CDU
Data 24/09/1998

Tese submetida para satisfação parcial

dos requisitos do programa de mestrado

em

Engenharia Electrotécnica e de Computadores

(Informática Industrial)

DEEC

FEUP

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Departamento de Engenharia Electrotécnica e de Computadores

PORTO

FEVEREIRO 1991

tese realizada sob a supervisão do

Doutor Eng. Eugénio da Costa Oliveira

Professor Auxiliar do
Departamento de Engenharia Electrotécnica e de Computadores
FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Quero expressar a minha gratidão a todos os que, por várias formas, contribuiram para a realização deste trabalho. De entre eles é justo destacar:

O Prof. Eugénio Oliveira, pelos conselhos e colaboração que me deu como orientador deste projecto, assim como pela disponibilidade com que sempre me recebeu.

A Texas Instruments - Equipamento Electrónico Portugal, pela disponibilização de tempo e recursos computacionais, e de servir de prova de ensaio para este trabalho.

Os meus colegas do Núcleo de I.A., dos quais devo destacar o colega Alcínio Miguel, pelo apoio e ambiente de trabalho que proporcionaram.

Os meus pais, pelo apoio e ânimo nos momentos mais difíceis.

A minha esposa, pelas longas horas sem a minha presença, pela revisão do texto, e principalmente pela força e vontade transmitida.

indocti discant et ament meminisse periti

Índice

	Preâmbulo	9
	Introdução	10
I	Resumo	11
I.1	Objectivo	11
I.2	Descrição da situação	15
I.3	Resumo dos resultados do programa	16
II	Fundamentos teóricos	18
II.1	Programação orientada por objectos	18
II.1.1	O mundo e os seus objectos	18
II.1.2	O que são objectos?	20
II.1.3	Que tipo de objectos podem ser descritos?	21
II.1.4	Como é que os objectos comunicam e se comportam?	21
II.1.5	Como estão organizados os objectos e seus métodos?	23
II.2	Smalltalk	26
II.2.1	Introdução	26
II.2.2	Os princípios básicos	27
II.2.3	A implementação	28
II.3	Simulação	29
II.3.1	Introdução à simulação	29
II.3.1.1	Origens e fundamentos	29
II.3.1.2	Vantagens e inconvenientes	30
II.3.1.2.1	Vantagens	30
II.3.1.2.2	Inconvenientes	30
II.3.1.3	A simulação-técnica e a simulação-método	31
II.3.2	Introdução à simulação-método	31
II.3.3	Método dos acontecimentos	33
II.3.3.1	Exemplo simples	33
II.3.3.2	Relógio	36
II.3.3.3	Medidas de eficiência do sistema	37
II.3.3.4	Caracterização das entidades	38
II.4	Ambientes industriais	39
II.4.1	Introdução	39
II.4.2	Equipamentos de produção	43

II.4.3	Fabrico de circuitos integrados	45
II.4.3.1	Introdução	45
II.4.3.2	Fabrico de circuitos integrados	46
II.4.3.3	Descrição de um FCC	47
II.4.3.3.1	*Assembly*	48
II.4.3.3.2	*Back-end*	48
II.4.3.3.3	*Test*	49
II.5	Estatística	50
II.5.1	Noções básicas de simulação-técnica	50
II.5.2	Números aleatórios	50
II.5.3	Simulação de variáveis aleatórias	51
II.5.4	Distribuições de probabilidade	52
III	Simulador de ambientes industriais - SimAI	53
III.1	Introdução	53
III.2	Distribuições de probabilidade	57
III.2.1	Distribuições discretas	58
III.2.1.1	Distribuição de Bernoulli	59
III.2.1.2	Distribuição Binomial	59
III.2.1.3	Distribuição Geométrica	59
III.2.1.4	Distribuição de Poisson	60
III.2.2	Distribuições contínuas	60
III.2.2.1	Distribuição Uniforme	60
III.2.2.2	Distribuição Exponencial	61
III.2.2.3	Distribuição Gamma	61
III.2.2.4	Distribuição Normal ou de Gauss	61
III.3	Estrutura base de simulação	62
III.3.1	Os objectos de uma simulação	62
III.3.2	Simulações	64
III.3.3	Implementação das classes do simulador	65
III.3.3.1	Classe SimulationObject	65
III.3.3.2	Classe DelayedEvent	67
III.3.3.3	Classe Simulation	68
III.4	Recolha de estatísticas	70
III.4.1	Estatísticas de duração	71
III.4.2	Histogramas de frequência	71
III.4.3	Contabilização de acontecimentos	72
III.4.4	Monitorização de acontecimentos	72

III.5	Recursos	74
III.5.1	Implementação das classes necessárias	75
III.5.1.1	A utilização de recursos	75
III.5.1.2	Coordenação de recursos	77
III.6	Interface com o utilizador	79
III.6.1	Controlo do programa	79
III.6.2	Écrans de entrada	79
III.6.3	Écrans de saída	80
III.6.4	Objectos essenciais	81
III.6.5	Estrutura do simulador	82
III.7	Exemplo de aplicação	83
III.7.1	Introdução	83
III.7.2	Descrição	84
III.7.3	Validação dos resultados	87
III.7.4	Outras abordagens	88
III.7.4.1	Folhas de cálculo	88
III.7.4.2	Sistemas periciais	89
III.7.4.3	Outros sistemas simuladores	89
III.7.4.4	Comparação com abordagens alternativas	91
IV	Conclusões	93
IV.1	Melhorias futuras	94
	Bibliografia	96
	Anexos	98
A.	Smalltalk	99
A.1	Um pouco de história	99
A.2	Smalltalk e outras linguagens convencionais	100
A.2.1	Afectação a uma variável escalar	100
A.2.2	Uma sequência de expressões	100
A.2.3	Chamar uma função com um argumento	100
A.2.4	Chamar uma função com dois argumentos	101
A.2.5	Chamar uma função com três argumentos	101
A.2.6	Acesso a variáveis indexadas	101
A.2.7	Expressões condicionais	102
A.2.8	Expressões iteractivas	102
A.2.9	Retorno de valores em funções	103

A.2.10	Afectação e recuperação de espaço de memória	103
A.2.11	Exemplo de um programa	103
B.	Estatística	106
B.1	Definições	106
B.2	Distribuições de probabilidade	107
B.2.1	Distribuições discretas	107
B.2.1.1	Distribuição de Bernoulli	108
B.2.1.2	Distribuição Binomial	108
B.2.1.3	Distribuição Geométrica	109
B.2.1.4	Distribuição de Poisson	109
B.2.2	Distribuições contínuas	110
B.2.2.1	Distribuição Uniforme	111
B.2.2.2	Distribuição Exponencial	111
B.2.2.3	Distribuição Gamma	112
B.2.2.4	Distribuição Normal ou de Gauss	112
C.	Montagem e teste de circuitos integrados	113
C.1	*Assembly*	113
C.1.1	*Die-prep*	113
C.1.2	*Mount*	114
C.1.3	*Bond*	115
C.2	*Back-end*	115
C.2.1	*Mold*	116
C.2.2	*Trim-Form*	116
C.2.3	*Symbol*	117
C.2.4	*Singulate*	118
C.3	*Test*	118
C.3.1	*V/M*	119
C.3.2	*Test* propriamente dito	119
C.3.3	*Pack*	120
D.	Algumas classes da simulação	121

Preâmbulo

Esta tese destina-se à obtenção do grau de mestre em Engenharia Electrotécnica e Computadores e vem no seguimento de uma parte lectiva em que concluí as seguintes cadeiras, que excedem os créditos mínimos necessários:

- Complementos de Matemática e Fisíca;
- Sistemas Informáticos e Métodos Numéricos;
- Economia e Gestão;
- Estruturas de Informação e Algoritmos;
- Projecto e Implementação com Microcomputadores
- Sistemas Gráficos;
- Controlo de Sistemas por Computador;
- Redes de Computadores;
- Processamento e Análise de Imagem;
- Projecto Assistido por Computador;
- Simulação;
- Projecto de Informática Industrial.

O trabalho desenvolvido inclui-se no domínio da Informática Industrial, e consistiu na criação de um modelo representativo de um ambiente fabril e sua posterior simulação de forma a obter informações que auxiliem o gestor industrial na optimização de regras de produção e na quantificação do equipamento necessário para a realização de um determinado volume de encomendas.

A aplicação desenvolvida, denominada Simulação de Ambientes Industriais (SimAI), pode ser usada para variados casos, sem necessidade de qualquer alteração do programa, através da introdução e alteração de valores que são apresentados em tabelas, pelo programa ao utilizador.

O sistema SimAI foi implementado em Smalltalk, que é uma linguagem de programação orientada por objectos. Foi criado um ambiente que permite gerar facilmente outras simulações orientadas por acontecimentos.

Para teste e validação do modelo e aplicação desenvolvidas, o sistema SimAI foi aplicado a um caso real, mais concretamente na empresa Texas Instruments - Equipamento Electrónico Portugal, empresa que proporcionou uma bolsa de estudos para a conclusão desta tese.

Introdução

Este relatório encontra-se dividido essencialmente em três capítulos. Esses capítulos foram denominados resumo, fundamentação teórica e simulação de ambientes industriais.

No resumo, pretendo dar uma visão geral do trabalho realizado. O objectivo é permitir um entendimento global de todo o trabalho com uma leitura breve. Serão abordados essencialmente os objectivos e os resultados e não os detalhes de implementação.

Na fundamentação teórica, pretendo dar uma visão dos domínios de conhecimento em que se baseou este trabalho. A sua leitura é benéfica para quem procure um melhor entendimento do trabalho realizado. Várias definições e conceitos são introduzidos. O seu objectivo não é atingir um nível que permita ao leitor fazer simulações ou programar numa linguagem orientada por objectos, como o Smalltalk, mas tão só permitir uma melhor leitura e compreensão do programa realizado. Para um conhecimento mais profundo dos temas abordados recomendo a bibliografia que se apresenta no final, que serviu aliás para a preparação desta parte. Foi ainda aqui incluída, na secção referente a ambientes industriais, a criação do modelo a usar no trabalho.

Na simulação de ambientes industriais, pretendo especificar o domínio de aplicação do trabalho desenvolvido, assim como descrever algumas das partes mais importantes da aplicação SimAI.

Com esta aproximação procurei obter as vantagens de duas alternativas de exposição. Ou partir do geral, detalhando o particular de seguida, ou partir dos vários componentes explicando a sua integração no final. Da forma escolhida, o leitor começa por uma percepção geral do todo, especializando cada parte seguidamente, e no final vendo a aplicação integrada dos conceitos apresentados.

A terminologia técnica tem uma base anglo-saxónica muito forte. Sempre que possível tentei usar terminologia portuguesa. No entanto como precaução relativamente a alguma tradução menos feliz ou pelo menos não habitualmente usada, é apresentada também a terminologia anglo-saxónica entre parêntesis e/ou em itálico da primeira vez que um termo é apresentado. O texto será habitualmente apresentado na fonte Times, sendo apresentado em fonte Helvetica expressões do programa em Smalltalk.

I. Resumo

I.1 Objectivo

O relatório que a seguir se apresenta corresponde a um trabalho de investigação e desenvolvimento na área da informática industrial. A partir deste trabalho foi desenvolvida uma aplicação de simulação de ambientes industriais.

Na identificação das necessidades da indústria em termos informáticos, surge nos primeiros lugares a simulação. Isto acontece devido ao custo, tempo e riscos envolvidos para se alterar muitos dos processos industriais. A simulação permite a experimentação de novas situações de forma segura, barata e rápida, permitindo as técnicas mais recentes uma aproximação elevada com o sistema real.

Escolhi como caso de aplicação o fabrico de circuitos integrados, quer por trabalhar e conhecer bem esse ambiente, quer por ser considerado um dos casos mais complexos que existe. Ao fazer um programa que se aplica a um caso complexo, estarão logo considerados outros casos mais simples.

Um dos problemas principais a resolver é o da quantificação correcta dos equipamentos necessários para atender a uma dada procura do mercado. Isto acontece porque os equipamentos industriais têm habitualmente um custo elevado e são apenas construídos por encomenda, tendo assim um tempo de entrega grande.

Num mercado de concorrência, uma eficiente gestão económico-financeira é fundamental, e a compra de equipamentos antes ou depois da altura correcta tem impacto negativo para a empresa.

Métodos matemáticos, facilitados pelo recurso a folhas de cálculo que correm em microcomputadores, são usados na quantificação dos equipamentos. No entanto, especialmente nos casos de fabrico de vários produtos por máquinas flexíveis, e quando as máquinas em diferentes fases de fabrico têm capacidades de produção muito diferentes, o uso de folhas de cálculo revela-se demasiado complexo, obrigando a simplificações consideráveis. O recurso à simulação permite uma análise da situação e um melhor entendimento das reais necessidades de equipamento.

Para uma correcta escolha das ferramentas a utilizar, comecei por definir os objectivos que se pretendiam atingir:

- Permitir a simulação de um ambiente do tipo industrial com o objectivo principal de verificar se a capacidade instalada é suficiente para uma dada procura;
- Permitir ainda testar diferentes políticas de gestão de equipamentos e inventários;
- O sistema deve ser fácil de usar tendo uma interface com o utilizador agradável e intuitiva;
- O sistema deverá adaptar-se facilmente a novos casos de aplicação do mesmo âmbito;
- O custo de correr o sistema deverá ser compatível com a dimensão da maioria das empresas nacionais.

Para se realizar a simulação existem várias hipóteses habitualmente separadas em dois grupos: a utilização de um sistema simulador de âmbito geral ou a utilização de uma linguagem de programação para a resolução específica do sistema em causa.

A segunda hipótese foi rapidamente afastada pois apesar de resolver o caso específico apresentado, seria pouco flexível. Quer para a adaptação a outros ambientes industriais, quer quando existisse alteração de processos, seria necessário alterar o programa para o novo caso concreto.

Na utilização de um sistema simulador de âmbito geral surgem novamente duas hipóteses: a utilização de um sistema comercial já disponível ou o desenvolvimento de um novo sistema.

Sou da opinião que não se deve desenvolver software que já exista, ou seja, não há necessidade de reinventar a roda. Fiz por isso uma procura de sistemas existentes no mercado. Verifiquei que praticamente não existem sistemas com o objectivo apresentado. A utilização de simuladores é relativamente recente e tem sido objecto de desenvolvimento essencialmente nas áreas de aplicação técnica e menos na de gestão. Existem vários sistemas simuladores de circuitos integrados, circuitos impressos, análise de sinal, redes eléctricas, movimentação mecânica, etc. Muitos destes sistemas estão orientados para colaborarem com sistemas de CAD (*Computer Aided Design)* no auxílio ao projecto eléctrico e mecânico.

Existem alguns sistemas, embora muito menos que os anteriores, com aplicação no domínio da gestão. Com um objectivo semelhante ao pretendido apenas identifiquei um sistema de uma companhia americana CACI, que é o SIMFACTORY. Tal sistema tem como principal desvantagem o seu elevado custo (mais de US$ 30.000). Mesmo assim realizei uma análise do mesmo, verificando a sua falta de flexibilidade em alguns pontos.

Do analisado anteriormente e em consideração com os objectivos definidos decidiu-se avançar com o desenvolvimento de um sistema simulador de ambientes industriais.

A primeira fase consistiu na escolha de um suporte de trabalho. O suporte de trabalho poderia variar desde um simples microcomputador até computadores de grande porte, com sistemas operativos ou *shells* do tipo DOS, Unix, Windows, Finder, etc. Como um dos objectivos era que o sistema deveria poder ser usado pela maioria das empresas nacionais, isso implica um suporte de trabalho que seja acessível para as mesmas. O sistema mais acessível e difundido na indústria é o microcomputador usando o sistema operativo MS-DOS, pelo que foi a plataforma escolhida. Como o Windows 3.0 está a ter uma aceitação cada vez maior, o sistema a desenvolver deveria preferencialmente, mas não obrigatoriamente, poder correr dentro dessa *shell*. No entanto se possível (e veremos mais tarde que foi possível) o sistema deveria poder correr em outras plataformas com um mínimo de alterações.

De seguida era necessário escolher uma linguagem de programação. Existem vários tipos de linguagens. O Assembly estava posto de parte devido ao tempo necessário para se desenvolver e corrigir um sistema desta envergadura. Existem algumas linguagens com alguns mecanismos para a criação de simulações, como o SIMSCRIPT, o GPSS, o SUSY, e temos ainda todo o conjunto de linguagens de programação normais. Da minha experiência pessoal (que inclui linguagens como: vários Assembly, C, Pascal, Lisp, Basic, Cobol, RPG, Susy, etc) não me sentia orientado para a escolha de qualquer delas. Recorri à literatura especializada para verificar quais as tendências actuais no domínio do desenvolvimento de software. Parece existir uma forte opinião que para grande parte das aplicações o futuro está no seu desenvolvimento em linguagens orientadas por objectos ([B8], [B9], [B10], [B11], e outros). Este é um novo paradigma da programação que me atraiu, especialmente pelos seus princípios.

Existem várias linguagens do tipo de programação orientada por objectos. Alguns exemplos são o Smalltalk, C++, Eiffel e Actor. A escolha recaiu em

Smalltalk por ter sido a primeira linguagem com este tipo de programação e por ser uma daquelas em que essa filosofia está mais integrada. Verifiquei ainda que o Smalltalk tem sido usado para modelização, simulação, planeamento e criação de sistemas periciais por empresas a colaborar no programa ESPRIT, pelo que deveria ser a decisão acertada.

Existem actualmente duas versões principais de Smalltalk: o Smalltalk-80 da ParcPlace Systems, e o Smalltalk/V da Digitalk Inc. Fiz uma análise de ambos e optei pelo Smalltalk-80 por ser aquele que responde melhor aos requisitos anteriormente definidos. O Smalltalk-80 permite correr o sistema desenvolvido em plataformas muito diferentes, sem qualquer alteração ao código. Alguns exemplos são computadores com MS-DOS, Macintosh, Sun e sistemas Unix. Permite também correr dentro da *shell* Windows 3.0. Tal não acontece com o Smalltalk/V, que necessita de pequenas alterações entre vários sistemas e que apenas tem versões para sistemas com MS-DOS e OS/2. O smalltalk-80 tem como desvantagem relativamente ao Smalltalk/V o facto de necessitar de (no caso de MS-DOS) microcomputadores no mínimo com o processador i386, coprocessador i387 e um mínimo de 4 MBytes (8 MBytes recomendado) de memória central. No entanto, este tipo de computadores está cada vez mais difundido no mercado pelo que é um requisito aceitável.

Do exposto anteriormente, a aplicação foi desenvolvida em:

- Microcomputador TI SP1000E: processador i386 mais coprocessador i387, 16 MHz, 4 MB, monitor VGA
- Sistema operativo: MS-DOS 3.3 mais Windows 3.0
- Linguagem de desenvolvimento: Objectworks for Smalltalk-80 v/2.5

A aplicação desenvolvida recebeu o nome de SimAI, sigla de **Sim**ulador de **A**mbientes **I**ndustriais.

I.2 Descrição da situação

Os ambientes do tipo industrial são caracterizados pela existência de um elevado números de parâmetros, pelo que necessitam de uma gestão cuidadosa.

Dos vários tipos de fabrico existentes o sistema SimAI aplica-se essencialmente a sistemas de produção por lotes a partir de um plano de encomendas. Outros sistemas poderão ser reduzidos a este mediante certas adaptações.

Uma fábrica está habitualmente dividida em áreas de produção. Um dado produto percorre essas áreas segundo uma dada sequência, incorporando matéria prima e mão de obra em cada uma delas. Na fabricação por lotes, cada lote é constituído por um conjunto de produtos de características semelhantes. O grupo evolui no processo de fabrico de forma conjunta, isto é, uma dada peça apenas passa para a área de fabrico seguinte quando todas as peças do grupo tiverem sido processadas nessa área. Quando o lote completa o processo numa dada área, segue para uma zona de inventário onde espera a disponibilidade de equipamento na área seguinte.

É também habitual que uma fábrica não produza apenas um produto específico, mas sim um dado conjunto de produtos mais ou menos semelhante. Assim nem todos os produtos terão de passar por todas as áreas da fábrica, assim como a sequência por que passam nas áreas pode ser diferente para cada tipo de produto. Uma dada área pode ter equipamentos que possam ser usados em vários produtos, e outros específicos para um dado produto.

Do apontado, verificamos a grande complexidade existente no cálculo apropriado da quantidade de equipamentos necessários, em algumas das áreas da fábrica, se não mesmo em todas.

A produção é consequência de um dado plano de encomendas. Este plano é resultado da recepção de encomendas dos clientes para uma determinada data e também da previsão realizada pelo departamento comercial. Assim surge um plano de necessidades ao longo de um período ou períodos de tempo. Do conhecimento do tempo necessário para fabricar um produto, obtemos um plano de produção adequado de forma a termos os produtos prontos na altura pretendida, e não antes ou depois.

I.3 Resumo dos resultados do programa

O sistema SimAI permite obter a reacção do ambiente fabril a uma dada pressão de produção. Existem vários parâmetros que podem ser vigiados de acordo com a nossa vontade. Pela variação da quantidade de equipamentos em cada área podemos verificar as alterações de comportamento obtidas.

Os parâmetros principais a que se pode ter acesso são:

- Por tipo de equipamento ou por área de trabalho:
  - utilização;
  - tempo de processo;
  - tempo de espera;
  - inventário.

- Por tipo de produto ou subproduto:
  - tempo de processo;
  - tempo de espera.

Os resultados apresentados são úteis para o estudo de alternativas de gestão de produção e gestão de inventários, e levam a um maior conhecimento do comportamento do sistema instalado.

O sistema permite conhecer para uma dada produção relativa entre vários produtos, qual o tempo médio de fabrico para cada produto, o que permite lançar a encomenda no período mais apropriado apenas recorrendo a pequenas margens de segurança.

Do conhecimento da capacidade instalada podem ser realizadas análises de sensibilidade. A encomenda de novos equipamentos e/ou a venda dos actuais pode ser planeada com tempo e com uma segurança apreciável.

O sistema SimAI foi aplicado ao caso de uma fábrica de circuitos integrados para se verificar a sua eficiência e utilidade. Este caso de aplicação tem como vantagem a existência de um grande número de equipamentos, produtos e fluxos diferentes, pelo que seria um teste mais completo que a maioria dos ambientes a que o sistema se pode aplicar.

O sistema foi testado para o nível de encomendas previsto para os próximos anos, tendo detectado a necessidade de encomendar alguns equipamentos assim como a existência de equipamentos em excesso em algumas áreas.

A reacção foi muito positiva, especialmente por parte das pessoas encarregadas de fazer o planeamento estratégico e engenharia industrial, os principais utilizadores deste sistema.

As alterações recomendadas correspondem a uma maior generalização dos resultados obtidos. Para além da comprovação dos equipamentos necessários para uma dada produção esperada, pretende-se também saber a quantidade de pessoas e espaço necessários. Uma outra recomendação foi a utilização do sistema numa plataforma mais potente, como por exemplo um microcomputador com o processador i486, pois o sistema é um pouco lento.

II. Fundamentos teóricos

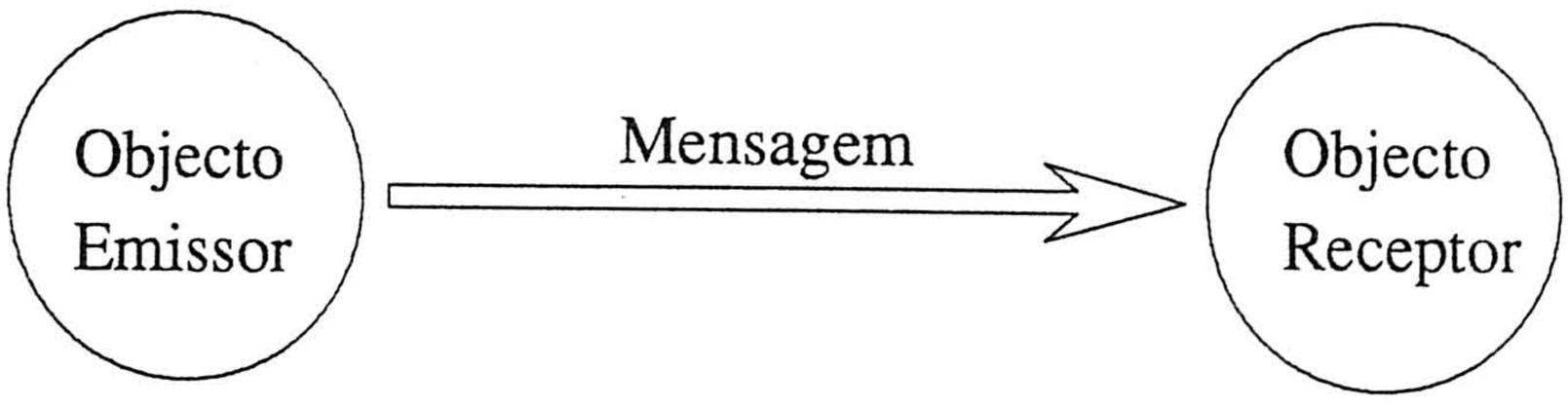
II.1 Programação orientada por objectos

A programação baseada em objectos é uma aproximação revolucionária em termos de abstração de dados, proporcionando uma nova forma de organizar os elementos de um sistema de software.

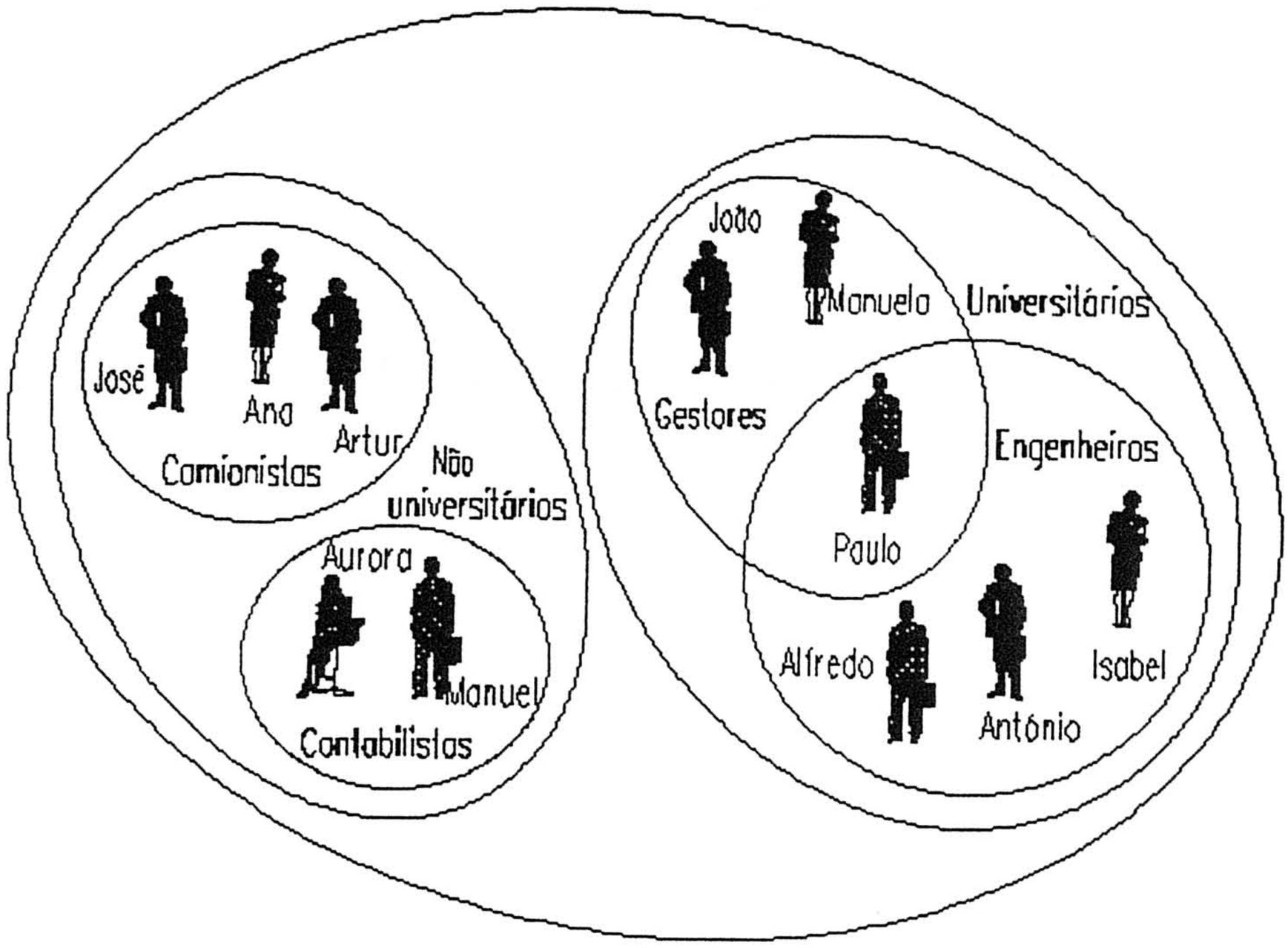
Vou apresentar uma introdução a este tema. Para maior desenvolvimento recomendo [B9], [B10], [B11] ou outras referências constantes na bibliografia

II.1.1 O mundo e os seus objectos

As linguagens de programação orientadas por objectos baseiam-se num modelo simples, mas muito poderoso de "objectos comunicantes". Esta é, habitualmente a aproximação mais natural à realidade. Nós sentimos o mundo que nos rodeia como um vasto conjunto de objectos discretos, agindo e reagindo num ambiente comum.

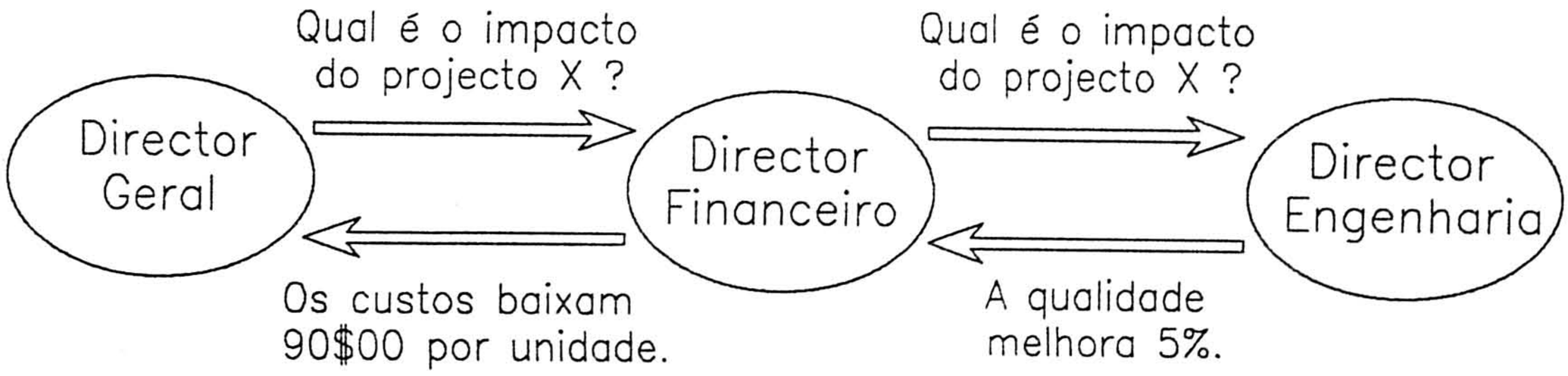


A nossa sociedade é constituída por engenheiros, médicos, advogados, contabilistas, padeiros, ladrões, etc. Embora sejamos uma população de indivíduos, podemos-nos associar em grupos de acordo com os conhecimentos e técnicas que cada um possui e demonstra.



Se partimos uma perna, chamamos um médico e partimos do princípio que ele sabe o que fazer para a tratar. Ou seja, nós (*self*) comunicamos com o médico (que é visto como uma caixa negra).

Se queremos ser advogados, o que temos a fazer é aprender Direito e comportarmo-nos como tal. Quando um director geral de uma companhia pergunta "Qual é a sua opinião sobre este projecto?", ele espera uma resposta diferente do advogado da companhia, do director financeiro ou do director de engenharia.



Em terminologia de programação orientada por objectos, ocupações em termos abstratos como médico, advogado, programador, etc, são classes (*classes*) das quais os indivíduos são instâncias (*instances*). Para nos tornarmos um

advogado, aprendemos os métodos (*methods*) das leis. A comunicação entre indivíduos é feita por mensagens (*messages*), cujo equivalente em termos de linguagem de programação são os selectores (*selectors*). É a correspondência entre a nossa percepção do mundo e a sua representação em termos máquina que está no centro do poder das linguagens orientadas por objectos.

II.1.2 O que são objectos?

Um objecto (*object*) é um conjunto relacionado de código e informação. As partes do código são os métodos, i.e. um conjunto de rotinas único para cada classe dando a essa classe o seu comportamento específico.. A estrutura de informação do objecto é descrita pelo agrupamento de variáveis da instância (*instance variables*).

Quando criamos uma determinada instância de uma classe, afectamos os valores iniciais das variáveis da instância. Os métodos dos objectos constituem o seu conhecimento. Se por exemplo pretendermos criar o objecto condutor de automóvel, ele teria de incluir métodos para travar, acelerar, meter mudanças, vigiar o tráfego, etc. As variáveis de instância de tal objecto poderiam incluir tempo de reacção, temperamento, capacidade visual, etc.

Um dado objecto contém programa e dados, comportando-se para o exterior como uma caixa negra relativamente ao seu interior. Essa caixa negra pode enviar e receber certas mensagens. A passagem de mensagens é a única forma de passar informação para o interior da caixa negra. Se um objecto necessita de fazer algo que não sabe, envia uma mensagem para outro objecto, pedindo ajuda na execução da sua tarefa.

Os objectos comunicam com outros objectos da mesma forma que advogados comunicam com contabilistas na analogia apresentada. O conhecimento profissional é comparável à colecção de métodos de um objecto. São as pessoas que comunicam usando o seu conhecimento e não o conhecimento de um que comunica com o conhecimento do outro. O conhecimento de um advogado ao preparar a defesa do seu cliente não possui uma ligação directa ao conhecimento do contabilista do seu cliente de forma a obter certas informações necessárias. Da mesma forma, os métodos de um objecto não chamam directamente métodos de outro objecto. Em vez disso os métodos (assim como o advogado) contêm

conhecimento de quando deverão pedir informações a outro objecto (neste caso o contabilista).

Em terminologia orientada por objectos, o encapsulamento da informação (*information hiding*), que é como é conhecida esta união de dados e programa, torna o software altamente portátil entre diferentes sistemas, facilmente modificável e seguro. Grandes aplicações podem ser facilmente mantidas, pois os objectos podem ser alterados, recompilados, testados e chamados imediatamente com as suas novas capacidades.

II.1.3 Que tipo de objectos podem ser descritos?

Tal como os seus equivalentes físicos, os objectos têm atributos e exibem comportamentos. Como tudo são objectos, incluindo o ambiente de desenvolvimento, o que se torna possível de programar numa linguagem é apenas uma questão de quais os objectos que podem ser descritos e manipulados.

Se o encapsulamento de informação proporciona meios para criar objectos, é a capacidade de abstração de informação da linguagem que determina os objectos que podem ser descritos. As linguagens orientadas por objectos, e o Smalltalk em particular, proporcionam essa capacidade de extender a sua própria linguagem permitindo criar novas estruturas de informação e modificar todo o ambiente de acordo com a nossa vontade. Depende da nossa própria capacidade de abstração o maior ou menor sucesso nessa tarefa.

II.1.4 Como é que os objectos comunicam e se comportam?

Os objectos tomam a responsabilidade pelas suas acções, respondendo eles próprios a todas as mensagens. A aplicação a desenvolver poderá ter de imprimir um número inteiro, um real ou um conjunto de caracteres. Cada um destes tipos elementares de dados é definido como uma classe, cujas instâncias contêm um determinado conjunto de características, os seus métodos. Cada um dos tipos elementares de dados sabe como se deve comportar em diferentes situações, como por exemplo nas operações de imprimir, duplicar, ou comparar.

Assim quando é necessário imprimir, a aplicação apenas necessita de enviar a mensagem de imprimir, que no caso particular do Smalltalk é printString, ao objecto que pretendemos, como por exemplo:

```
'Isto é uma frase' printString.
423 printString.
#(1 2 3) printString.
$A printString.
#('array de' 3 'strings e' 2 'números') printString.
```

Os inteiros, matrizes ou caracteres tomam a responsabilidade de se imprimirem de forma correcta no papel. Esta característica das linguagens orientadas por objectos de responderem de forma adequada a uma mesma mensagem é conhecida por polimorfismo (*polymorphism*). Isto significa não termos de memorizar um vocabulário específico para cada classe a usar nas nossas aplicações.

Como são os próprios objectos que tomam a responsabilidade do seu comportamento, não é necessário encher a nossa aplicação de verificações condicionais para verificar se estamos a usar o tipo apropriado de função para operar num determinado tipo de dados. Esta característica poupa muito tempo e reduz de forma significativa o custo de manutenção de software, pois apenas os objectos afectados necessitam de ser alterados e recompilados ao melhorar uma dada aplicação.

Entre as muitas razões que levam os objectos a comunicar, um objectivo frequente é alterar o estado do objecto receptor. Enviar uma mensagem para colocar um novo valor numa variável (uma variável é um objecto que guarda outros objectos), é um exemplo de alteração de estado por mensagens. Um outro exemplo, é o pedido feito pelo utilizador para alterar o tamanho de uma dada janela. O resultado da interacção com o movimento do rato (*mouse*) envia novas coordenadas para as variáveis da instância dessa janela que guardam a nova localização e tamanho no écran. O estado da janela muda pela recepção da mensagem de alteração de tamanho.

É sempre garantido que haverá uma resposta pelo receptor da mensagem. Se um objecto decide que não sabe como reagir a um dado pedido, ele responderá no mínimo com a mensagem "Mensagem não entendida" (*"Message not understood"*). O método que responde desta forma requisita os serviços de depuração de erros do

sistema (*debugging utilities*) de forma a ajudar o utilizador a descobrir e corrigir a falha de comunicação.

Assim até os erros do programa podem ser detectados e corrigidos dentro do ambiente orientado por objectos através da troca de mensagens. Isto proporciona um ambiente exploratório para desenvolvimento de programas. Um protótipo pode ser rapidamente criado e melhorias serem acrescentadas facilmente ao sistema.

II.1.5 Como estão organizados os objectos e seus métodos?

As classes estão organizadas de uma forma hierárquica de classes e subclasses. Por exemplo, na linguagem Smalltalk, a classe Integer é uma subclasse da classe Number, que por sua vez é uma subclasse da classe Magnitude, que ainda é uma subclasse da classe Object, a classe mais geral de todas e raíz de todas as outras classes.

A herança de características (*inheritance*) proporciona um mecanismo para organizar e manter um dado conjunto de classes. Ela reconhece semelhanças entre objectos, realçando o facto de que objectos semelhantes também reagem de forma semelhante.

Uma subclasse herda todos os métodos conhecidos do seu "progenitor". Por exemplo se definirmos uma nova classe que é uma base de dados de clientes como subclasse de SortedCollection (um conjunto ordenado de dados em Smalltalk), esta classe saberá automaticamente como adicionar, copiar, alterar, remover, imprimir ou ordenar os registos dos clientes. Apenas teremos de criar os métodos que a base de dados terá de conhecer para além daqueles já conhecidos da classe genérica SortedCollection.

Se um determinado comportamento herdado não é desejado, apenas teremos de definir um novo método com o mesmo nome do herdado que actue da forma pretendida. Por exemplo para proporcionar maior segurança de forma a proteger informações dos nossos clientes, a nossa classe poderia ser escrita de forma a pedir um código antes de imprimir.

A "herança" de comportamentos encoraja um estilo de programação por excepção e através de modificações ao já existente. Sem esta característica

teríamos de perder muito tempo a especificar para cada classe de objectos como realizar operações elementares, como por exemplo imprimir. E se por acaso, no futuro fosse necessário alterar a forma de imprimir uma dada estrutura de informação, teríamos potencialmente centenas de métodos para alterar e recompilar.

Esta estrutura hierárquica de classes e as características referidas correspondem à forma como percebemos e classificamos o mundo que nos rodeia.

```
Object
    Behavior
        ClassDescription
            Class
            Metaclass
    BitBlt
        CharacterScanner
        Pen
    Boolean
        False
        True
    Collection
        Bag
        MappedCollection
        SequenceableCollection
            ArrayedCollection
                Array
                IntegerArray
                RunArray
                String
                Text
            Interval
            LinkedList
                Semaphore
            OrderedCollection
                SortedCollection
        Set
            Dictionary
                CEnvironment
                IdentityDictionary
                SystemDictionary
            IdentitySet
                SignalCollection
    CompilerErrorHandler
    Controller
    ControlManager
    Delay
    DisplayObject
        DisplayMedium
            Form
                Cursor
                DisplayScreen
        DisplayText
            Paragraph
                TextList
        InfiniteForm
        OpaqueForm
        Path
            Arc
                Circle
            Curve
            Line
            LinearFit
            Spline
    Filename
    FormBitmap
        DisplayBitmap
    Link
        Process
```

```
Object
    Magnitude
        ArithmeticValue
            Number
                Fraction
                Integer
                    LargeNegativeInteger
                    LargePositiveInteger
                    SmallInteger
                LimitedPrecisionReal
                    Float
            Point
        Character
        Date
        LookupKey
            Association
        Time
    MenuBuilder
    Message
    Model
    PopUpMenu
        ActionMenu
        HierarchicalMenu
            PopUpTextMenu
    ProcessorScheduler
    Rectangle
    SharedQueue
    Signal
    SmalltalkCompiler
    Stream
        BinaryStorage
        PeekableStream
            PositionableStream
                ExternalStream
                InternalStream
        Random
    UndefinedObject
    VariableDefinition
    View
        BinaryChoiceView
        DisplayTextView
        FormMenuView
        FormView
            FormHolderView
        IconView
        ListView
            ChangeListView
            SelectionInListView
        StandardSystemView
            BrowserView
            FileListView
            InspectorView
            NotifierView
    StringHolderView
            FillInTheBlankView
            ProjectView
            TextCollectorView
    SwitchView
            BooleanView
    TextView
            CodeView
```

Visão parcial da hierarquia de classes do Smalltalk-80

II.2 Smalltalk

II.2.1 Introdução

O Smalltalk é uma linguagem e um ambiente de desenvolvimento caracterizado por um estilo puro de programação orientada por objectos, com todas as vantagens que daí advêm. Foi ele aliás o grande precursor deste estilo de programação.

Para um conhecimento mais aprofundado desta linguagem recomendo [B12], [B16], [B17] ou outras referências citadas na bibliografia. Um resumo da origem do Smalltalk e a sua comparação com a linguagem Pascal pode consultar-se no anexo A.

Já foram criadas muitas aplicações em Smalltalk, em domínios como a simulação, sistemas periciais, tutores inteligentes, interfaces de inquérito a bases de dados, sistemas informáticos integrados, etc.

Isto acontece porque o Smalltalk é ao mesmo tempo uma linguagem de programação muito poderosa, sendo possíveis reacções complexas com apenas algumas linhas de código, e um poderoso ambiente de desenvolvimento, com vários utilitários e enorme quantidade de rotinas que se podem copiar e redefinir para usar nos nossos próprios programas.

O Smalltalk encoraja um estilo de programação baseado em "projecto - protótipo - melhoria", ou seja o desenvolvimento incremental de um programa. O Smalltalk permite a criação, alteração e instalação de pequenas quantidades de código sem longas esperas de compilação e "linkagem", criando um programa pouco a pouco e vendo os resultados imediatamente. Isto permite explorar ideias e algoritmos, assim como testar o software muito antes de todo o programa estar completo.

Exceptuando um pequeno núcleo escrito em linguagem máquina, o Smalltalk é escrito em Smalltalk. O código (*source*) é fornecido de forma comentada, pelo que pode ser usado quer para modificar o sistema, quer para se copiar parte dele para nossa utilização depois de adaptado.

Será feita habitualmente referência a Smalltalk e não a qualquer das suas implementações comerciais. Existem fundamentalmente duas implementações comerciais de Smalltalk. O Objectworks/Smalltalk-80 da ParcPlace Systems é o sistema original e é aquele em que foi desenvolvido este trabalho. Este sistema corre em diferentes sistemas operativos como MS-DOS, Finder, Unix, Sun, etc, com elevado nível de compatibilidade entre os vários sistemas. Outra versão baseada na anterior é o Smalltalk/V da Digitalk Inc. que existe em versões diferentes respectivamente para os microprocessadores Intel 8088, 80286 e para o Presentation Manager sob o OS/2.

II.2.2 Os princípios básicos

O Smalltalk baseou-se num pequeno conjunto de ideias poderosas:

- O principal componente de um sistema informático é o utilizador.
- Programar deve ser uma extensão natural do acto de pensar.
- Programar deve ser um processo evolutivo de acordo com o modelo de aprendizagem que os seres humanos possuem.
- Um ambiente de programação é ao mesmo tempo uma linguagem e uma interface de ferramentas poderosas para programadores/ /utilizadores, que permitam uma fácil expressão nessa linguagem organizando e usando de forma flexível conhecimento procedimental e factual.

O Smalltalk incorpora estas ideias numa estrutura de comunicação do tipo ser humano/computador. Na sua forma mais simples, o Smalltalk é mais uma linguagem de programação como o Basic, C, Pascal ou Prolog.

O sistema é seguro de forma que é muito difícil provocar um bloqueio (*crash*) do sistema. Os erros em Smalltalk são apenas objectos que nos comunicam que não percebem aquilo que lhes estamos a pedir. A criação de módulos de dados e programas característicos deste estilo de programação favorece uma táctica de "dividir para conquistar", na resolução dos vários problemas de programação.

Os objectos em Smalltalk são fáceis de inspecionar, duplicar, modificar e talvez o mais importante, reutilizar. Isto permite que se use o tempo disponível na resolução do problema e não a voltar a escrever aquilo que já foi desenvolvido.

Os termos básicos usados em Smalltalk são apenas cinco: objecto, mensagem, classe, instância e método. Estes termos já foram introduzidos na secção anterior.

II.2.3 A implementação

O sistema do Smalltalk-80 pode ser dividido em dois componentes principais: a imagem virtual e a máquina virtual. A imagem virtual consiste no conjunto de todos os objectos do sistema. A máquina virtual consiste na interface com o hardware e nas rotinas de código máquina que permitem as relações entre os objectos na imagem virtual.

A imagem virtual parte do princípio que existem as seguintes condições:

- Um écran do tipo bitmap (de preferência de elevada resolução);
- Um dispositivo apontador com três botões (embora seja possível a utilização de dispositivos de dois ou um botão);
- Um teclado;
- Um contador de milisegundos;
- Um relógio de tempo real com uma resolução mínima de um segundo.

Para a implementação do sistema num dado hardware é apenas necessário desenvolver a máquina virtual. Após isto, pode ser carregada a imagem virtual que contém o fulcro do sistema, e que é independente do hardware em que se encontra instalada.

Os métodos escritos em Smalltalk são expressões representadas como instâncias da classe String. Estas expressões contêm uma sequência de caracteres compatível com a sintaxe da linguagem. Da primeira vez que um método é invocado o compilador do sistema faz a sua tradução para instruções de oito *bits* denominadas *bytecode* orientadas para utilização numa máquina de *stack*. Estas instruções são então interpretadas sempre que necessário.

Um programador ou utilizador não actua directamente com o compilador ou o interpretador. Toda esta fase é transparente.

Caso um utilizador pretenda acrescentar parte de um programa escrito noutra linguagem, como por exemplo C, tal é possível.

II.3 Simulação

II.3.1 Introdução à simulação

Uma análise mais aprofundada deste tema pode ser feita em [B1], [B2], [B3], [B4], [B5], [B6] ou [B7].

II.3.1.1 Origens e fundamentos

Há muitos problemas de decisão que, devido à sua estrutura interna ou à sua complexidade, são muito difíceis (ou impossíveis) de analisar com os métodos mais tradicionais de criação de modelos, como por exemplo o método dedutivo e o de inferência estatística. Esta dificuldade é resultado da difícil explicitação de uma função que represente o modelo. Nestes casos (mas não unicamente) o recurso à simulação é recomendável e quase inevitável.

O objectivo da simulação é imitar o comportamento do sistema através de um modelo capaz de:

- Reproduzir a sucessão de estados pelos quais o sistema passa ou pode passar;
- Registar medidas relevantes na definição da utilidade ou eficiência do sistema.

Uma vez construído, o modelo é utilizado como banco de ensaio de diferentes políticas que, uma vez validadas, podem ser introduzidas no sistema real.

Um modelo de simulação pode tomar várias formas, por exemplo:

(i) Modelo físico a uma escala reduzida

Exemplos: Avião num túnel de vento;
Modelo de uma rede de caminho de ferro.

(ii) Modelo físico analógico

Exemplos: Tubos com água colorida são usados num modelo macro-económico do Reino Unido (London School of Economics).
Utilização de circuitos eléctricos para imitar o processo de ventilação de uma mina de carvão.

(iii) Modelo digital (normalmente dependentes do tempo)

É um conjunto de equações matemáticas e/ou relações lógicas resolvido num computador. Também podem ser resolvidos manualmente, mas é um processo extremamente fastidioso.

II.3.1.2 Vantagens e inconvenientes

II.3.1.2.1 Vantagens

(i) A simulação tem a vantagem de ser, pelo menos superficialmente, compreensível por parte do agente de decisão.

(ii) O próprio processo de construção do modelo obriga os agentes de decisão a especificar com precisão as interacções entre os vários componentes do sistema, o que em si pode levar a uma melhor compreensão do problema real.

(iii) Grande flexibilidade para analisar mudanças no sistema, mesmo que de carácter estrutural.

(iv) Evita elevados custos de experimentação até se chegar à solução mais eficiente.

(v) Permite avaliar com segurança as reacções de sistemas perigosos e/ou valiosos a várias situações.

II.3.1.2.2 Inconvenientes

(i) O resultado de um modelo de simulação apenas exprime uma relação pontual $\{x_i^o, y_i^o\} \rightarrow U^o$ e não permite a explicitação de $U = f(x_i, y_i)$.

(ii) Normalmente exige um grande esforço computacional, sobretudo na fase de construção do modelo.

II.3.1.3 A simulação-técnica e a simulação-método

Até aqui tem sido apenas referido a simulação-método, isto é, o processo de representar os vários componentes de um sistema e as suas interdependências, com o objectivo de imitar o seu comportamento e de analisar as consequências de uma dada política de acção.

Porém, a palavra simulação é também frequentemente utilizada para descrever o processo de geração e imitação do comportamento de variáveis aleatórias, e controlar os efeitos da sua utilização em modelos de simulação. Designa-se por simulação-técnica o conjunto de técnicas que permite a utilização de variáveis aleatórias em modelos de simulação. Estas técnicas serão abordadas na secção sobre estatística.

II.3.2 Introdução à simulação-método

Qualquer processo ou situação que pretendamos simular é considerada um sistema. Um sistema é um grupo de objectos interactuando de forma a produzir um dado resultado. Por exemplo, uma fábrica é constituída por um grupo de pessoas e máquinas trabalhando em conjunto para fabricar um dado produto ou série de produtos.

Um sistema pode ser discreto ou contínuo, dependendo da forma como o estado dos objectos varia no sistema. Um sistema contínuo tem parâmetros que podem tomar qualquer valor real num dado intervalo (por exemplo temperatura ou pressão). Num sistema discreto os parâmetros apenas podem tomar valores de entre um conjunto fixo de possibilidades (por exemplo o número de máquinas numa fábrica). A simulação de um dado sistema opera com parâmetros do mesmo tipo.

Um sistema pode ainda ser determinístico ou estocástico dependendo das relações entre os seus valores de entrada e de saída. Num sistema deterministico é possível prever a sua reacção desde que se conheça o seu estado inicial e todas as suas entradas. Num sistema verdadeiramente estocástico, não existe explicação para a sua aleatoriedade. Um sistema que requeira um elevado esforço para ser representado de forma determinística pode ser mais fácil de representar de forma estocástica. A maioria dos sistemas contém parâmetros estocásticos e

determinísticos, mas o efeito aleatório introduzido pelos parâmetros estocásticos faz com que o sistema apresente um comportamento desse tipo.

Para se realizar a simulação de um sistema, é necessário criar um modelo desse sistema. Um modelo é um conjunto de informação que representa o sistema de uma forma diferente, possível de ser tratado de forma teórica. Não existem modelos únicos, pois a forma do modelo depende da informação selecionada para representar o sistema. A criação de um modelo é a fase mais importante da simulação. Um modelo é sempre mais simples que a situação real que representa. É necessário escolher os parâmetros mais representativos do sistema e relacioná-los de forma correcta. A introdução de parâmetros de pouco impacto vai aumentar o esforço computacional de processamento do modelo sem introduzir alterações apreciáveis nos resultados.

O tempo é uma característica sempre presente numa simulação. Embora existam muitos processos de como processar esta variável numa simulação, vou apenas referir os dois mais comuns.

Numa simulação orientada pelo tempo (*time driven*) um relógio é usado para manter o registo do tempo da simulação. O relógio apenas incrementa o seu valor uma unidade de cada vez. Depois de cada incremento o programa deve inspecionar cada um dos acontecimentos para verificar se ele deve ocorrer naquele momento. Se o acontecimento deve ter lugar, a actividade associada com ele é executada. Esta aproximação é habitualmente usada em simulações contínuas e quando o número de acontecimentos em cada momento é elevado.

Um método mais eficiente, mas não tão óbvio, consiste na simulação orientada por acontecimentos (*event driven*). O programa deve inspecionar todos os acontecimentos para detectar qual deverá ser executado de seguida. Cada acontecimento guarda o valor de relógio onde deverá voltar a ser executado. Depois do relógio principal ser actualizado, a actividade associada com o acontecimento é processada e o acontecimento calcula (e guarda) o valor de relógio em que deverá novamente ser executado. Este método é o habitualmente usado em simulações discretas.

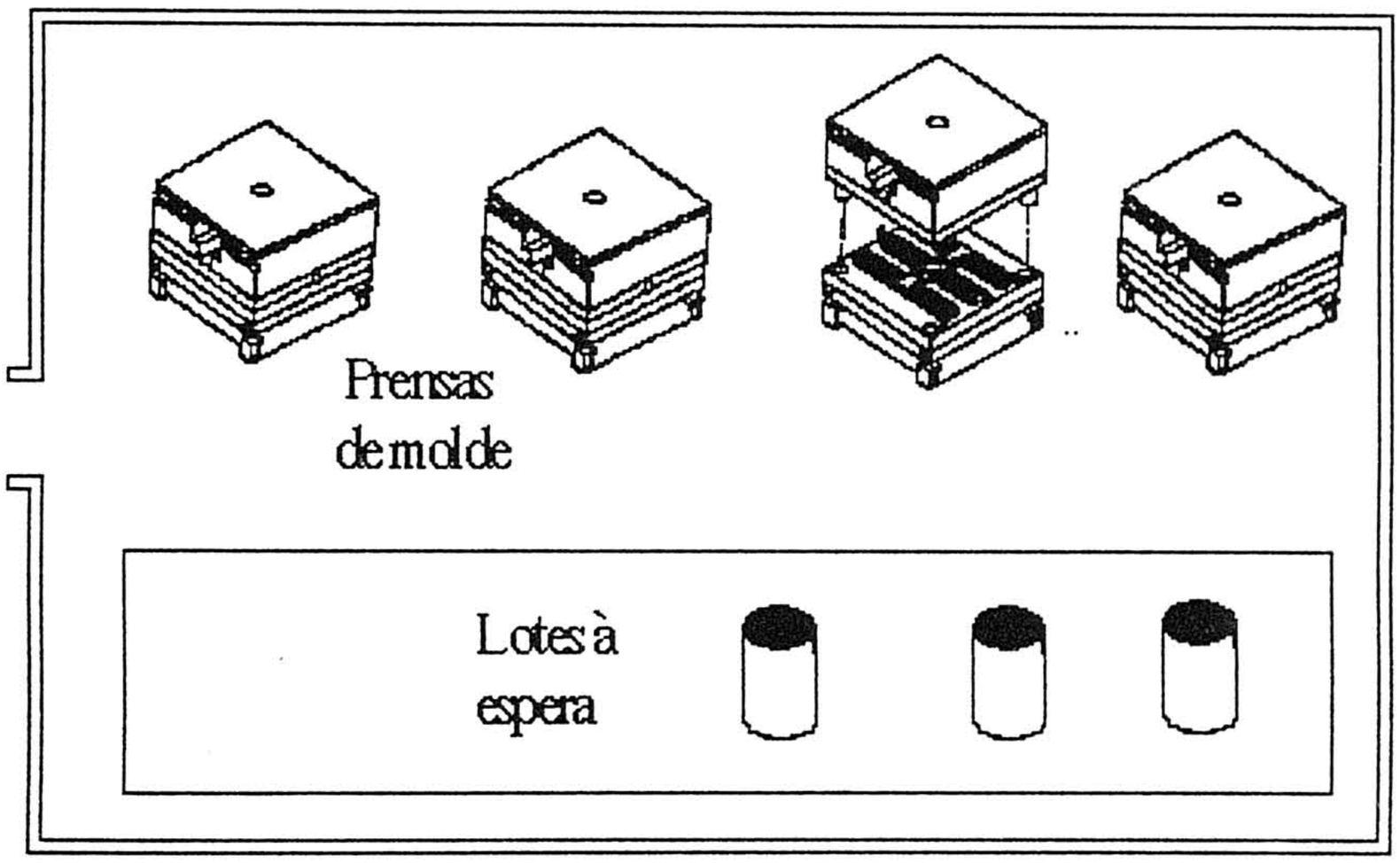
A simulação de ambientes industriais é um caso de simulação discreta pelo que o método que usei e que deverá ser depreendido no restante texto é o da simulação discreta orientada por acontecimentos.

II.3.3 Método dos acontecimentos

Este método baseia-se na modelização das mudanças de estado do sistema (acontecimentos).

II.3.3.1 Exemplo simples

Considere-se o problema de um director de produção que pretende estudar se vale a pena aumentar o número de equipamentos na área de moldagem



O primeiro passo para a construção do modelo de simulação é a identificação dos acontecimentos, isto é, as possíveis mudanças de estado do sistema.

Neste caso poderíamos identificar:

1. Chegada de um lote;
2. Início da moldagem;
3. Fim da moldagem e saída do lote para a área seguinte.

É evidente que o número e caracterização dos acontecimentos depende do detalhe que se pretende introduzir no modelo. No caso do exemplo que estamos a considerar poderíamos transformar o 3º acontecimento nos seguintes:

3.1 Fim da moldagem;
3.2 Início de retirar o lote da prensa;
3.3 Fim de retirar o lote e saída do mesmo.

Para além dos acontecimentos é ainda necessário identificar os seguintes elementos:

Entidades: são elementos (ou entes) de quem depende a mudança de estado do sistema;
Filas: são espaços físicos ou lógicos onde se colocam entidades e que têm associados critérios de selecção.

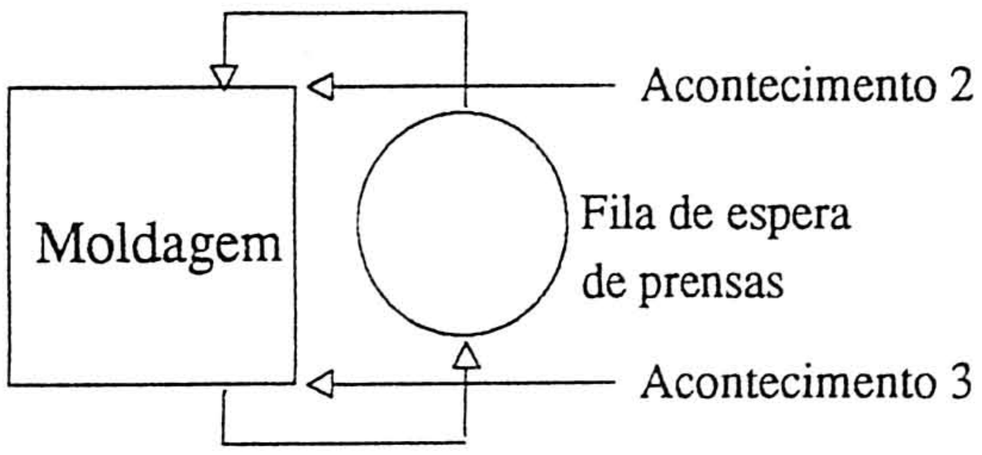
No exemplo da área de molde podemos identificar:

Entidades: Lotes e Prensas;
Filas: Inventário (lotes) e Prensas inactivas.

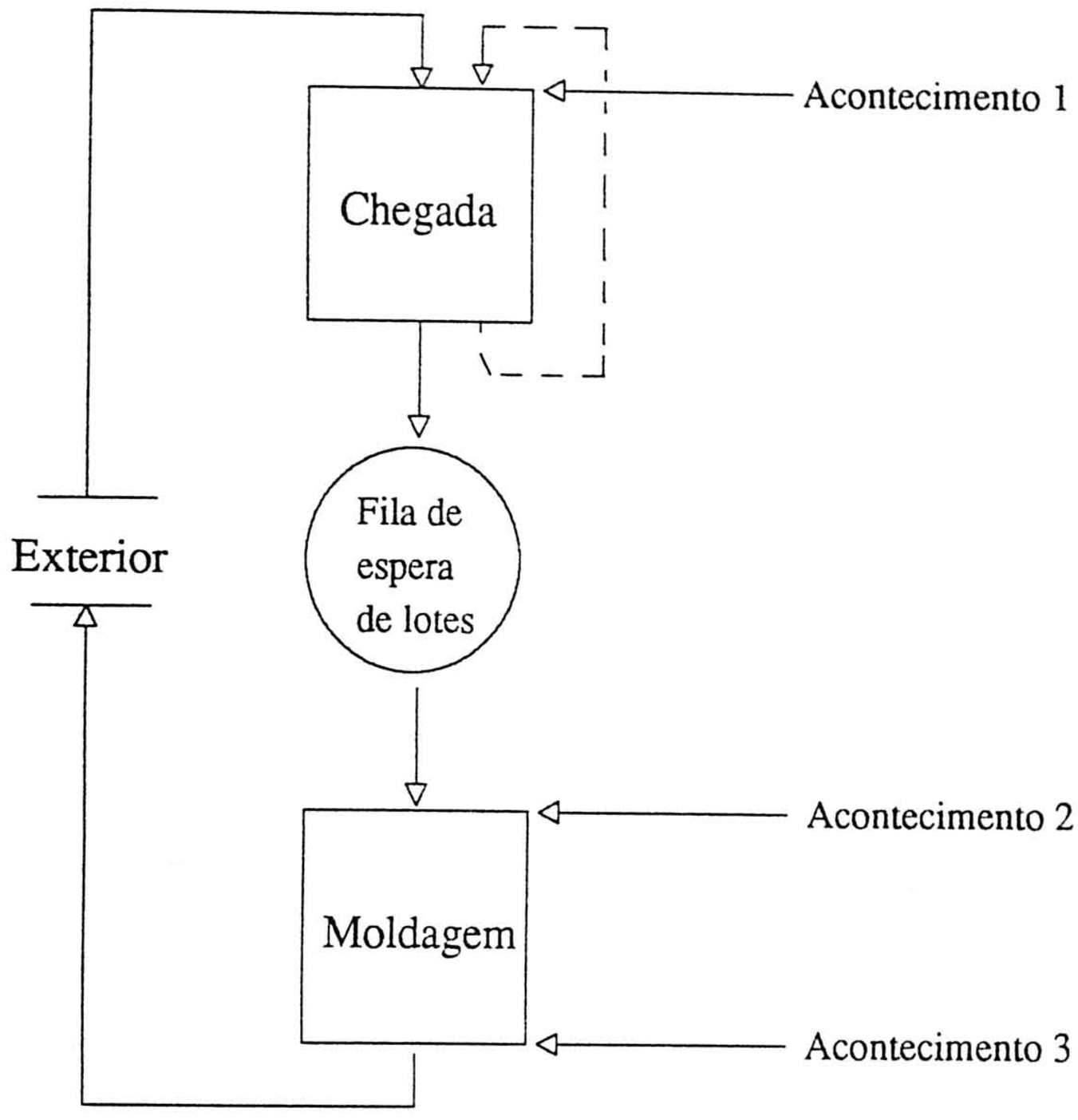
Quando existe mais do que uma entidade cujo comportamento no sistema é semelhante, definem-se grupos de entidades. No caso presente tanto os lotes como os prensas são grupos de entidades.

O passo seguinte na construção do modelo é a caracterização da sequência de acontecimentos e tempos inactivos de cada grupo de entidades.

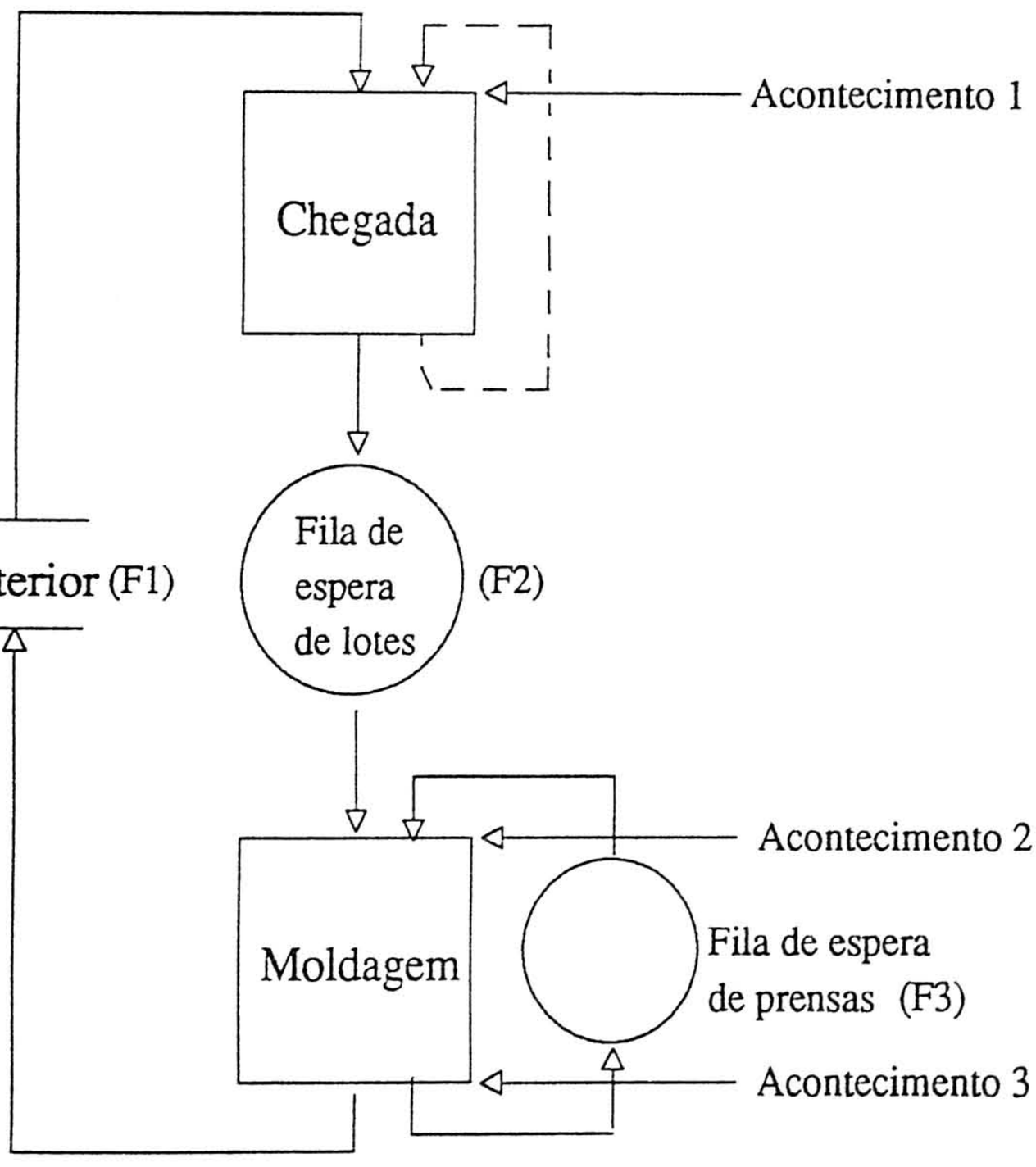
(i) Diagrama de sequência para prensas:



(ii) Diagrama de sequência para lotes:



(iii) Diagrama de sequência global:



Caracterização dos acontecimentos:

Acontecimento 1 - chegada de um lote

(i) Retirar o 1º lote de fora (F1)
(ii) Colocar lote no último lugar de F2
(iii) Marcar próxima chegada [A1] $\Delta T = t1$
(iv) Se # F2 = 1 e # F3 >= 1 então marcar [A2] $\Delta T = 0$

Acontecimento 2 - início da moldagem

[Condições: # F2 >= 1 e # F3 >= 1]
(i) Retirar o 1º lote de F2
(ii) Retirar a 1ª prensa de F3
(iii) Calcular a duração da moldagem (t_2)
(iv) Marcar [A_3] $\Delta T = t_2$

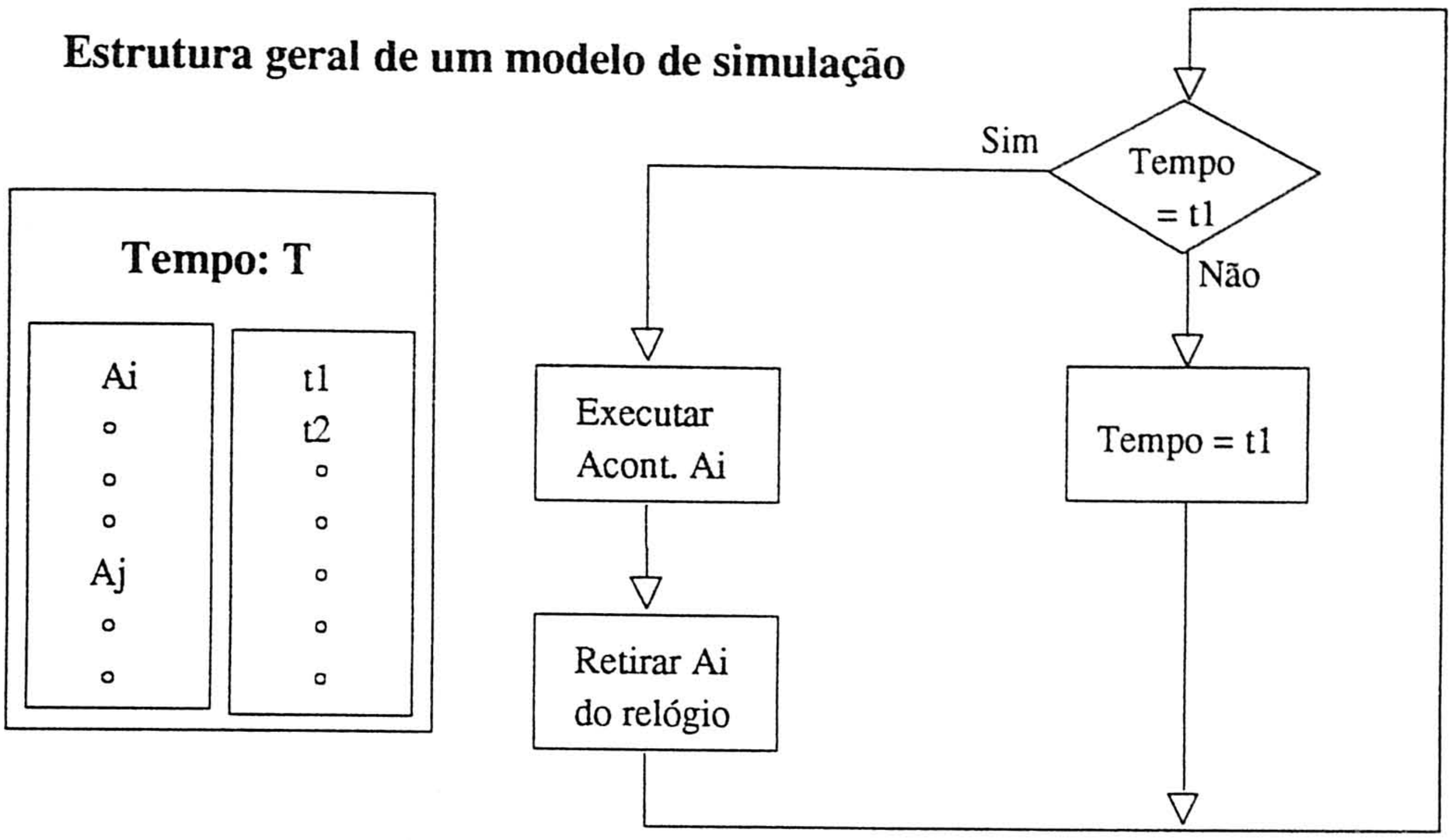
Acontecimento 3 - fim da moldagem e saída do lote

(i) Colocar lote em F1
(ii) Colocar prensa em F3
(iii) Se # F2 >= 1 marcar [A_2] $\Delta T = 0$

II.3.3.2 Relógio

O relógio é o elemento de qualquer modelo de simulação que controla a sua evolução.

Associado ao relógio existe uma fila de espera lógica onde os acontecimentos futuros estão ordenados por ordem decrescente do tempo da sua ocorrência.



II.3.3.3 Medidas de eficiência do sistema

Até aqui analisei como é possível descrever com o detalhe pretendido qualquer sistema cuja mudança de estado ocorra em instantes discretos. Porém é fundamental que sejamos capazes de avaliar a eficiência do sistema.

Por exemplo no exemplo anterior, está-se interessado em determinar qual é o número de prensas de molde a instalar.

Em regra geral, interessa-nos coligir informações que permitam avaliar a eficiência de configurações e/ou regras de funcionamento.

No caso da área de molde poderiam ser:

Entidades: Tempo médio de espera dos lotes;
Tempo máximo de espera dos lotes;
Taxa de utilização das prensas;
Filas: Histograma do número de lotes à espera.

II.3.3.4 Caracterização das entidades

Muito embora já tenha caracterizado o comportamento genérico das entidades através dos diagramas de sequência para cada grupo, por vezes é necessário distinguir algumas características.

Por exemplo no caso anterior, pode-se estar interessado em distinguir lotes diferentes segundo diferentes características:

(i) tipo de plástico;
(ii) com ou sem inspeção;
(iii) tamanho do molde.

Para ser capaz de fazer este tipo de caracterização mais específica usam-se atributos. Estes também são indispensáveis para armazenar, ao longo da execução do modelo, informações relevantes para a determinação de medidas de eficiência.

Por exemplo:

(i) hora de entrada de lote;
(ii) duração da moldagem.

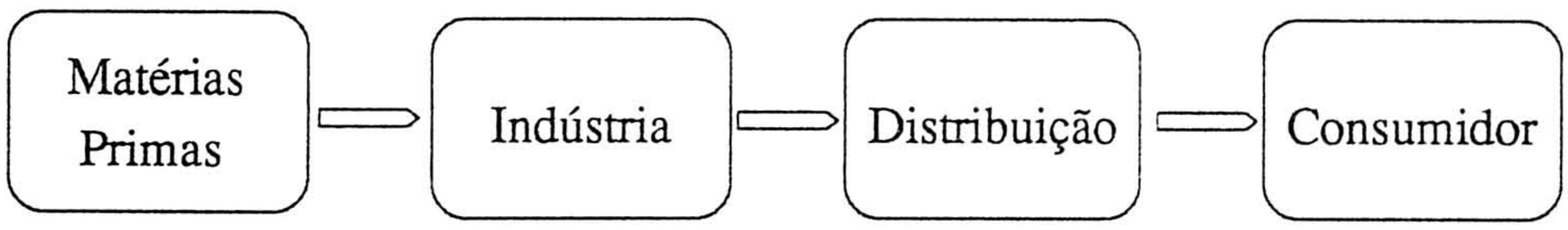
II.4 Ambientes industriais

II.4.1 Introdução

O sector industrial é um componente de elevada importância no sistema económico de qualquer país. É ele que produz os bens que utilizamos, assim como os equipamentos para produção de outros bens ou para a manutenção de toda a actividade de um país.

Em termos gerais o sector industrial adquire matérias primas, transforma-as num determinado produto, passando de seguida esses produtos ao sector de distribuição que o fará chegar aos consumidores finais.

O fabrico de um dado produto pode ser realizado por apenas um fabricante, ou ter necessidade de passar por vários fabricantes até ser usado pelo utilizador final. Por exemplo um produto electrónico, como um computador, é constituído por um número elevado de componentes electrónicos produzidos por diferentes fabricantes.



Uma empresa industrial, habitualmente designada neste trabalho por ambiente industrial, pode estar organizada de diferentes formas consoante o tipo dos produtos que fabrica, mas a estrutura genérica é suficientemente comum, possibilitando a criação de um modelo abstrato geral.

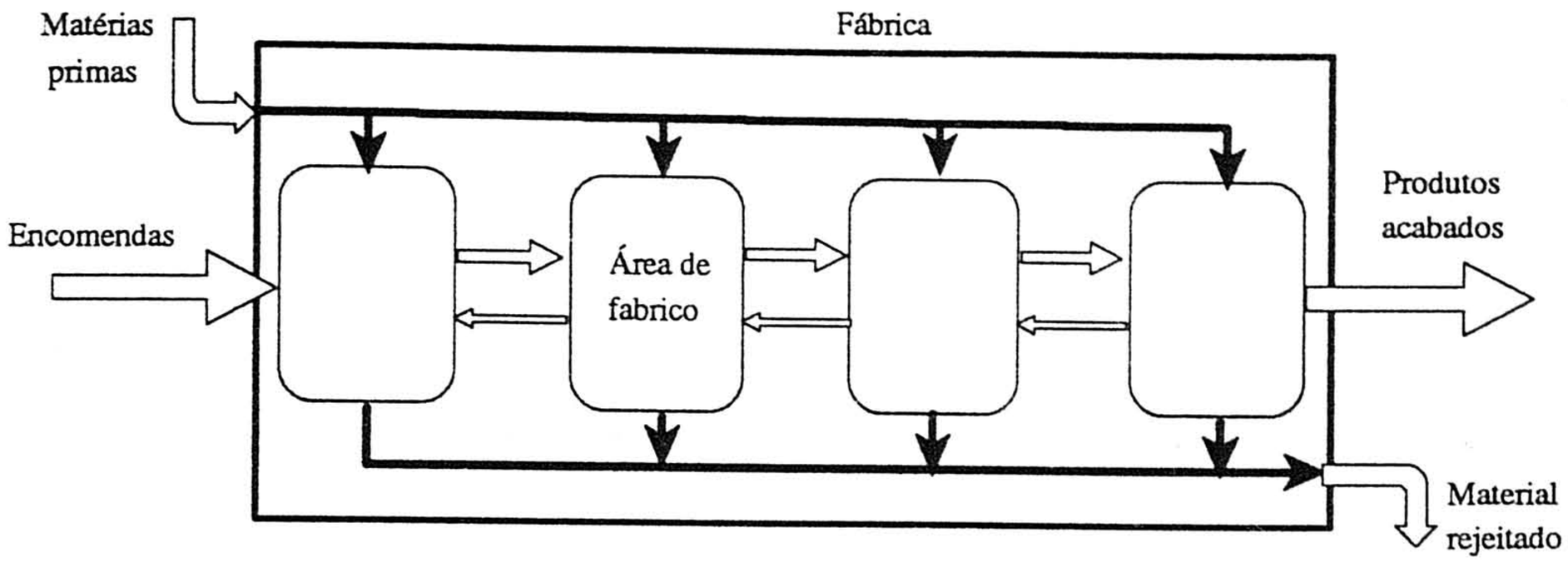
Um ambiente industrial pode ser dividido em várias áreas ou sectores de fabrico. Cada área é responsável por acrescentar algo ao produto em curso de fabrico. Como tipicamente uma fábrica tem diferentes produtos, não é necessário que um dado produto tenha de passar por todas as áreas da fábrica. Denomina-se como fluxo de um produto, a sequência ordenada de áreas (ou equipamentos) que um determinado produto deverá percorrer até estar concluído.

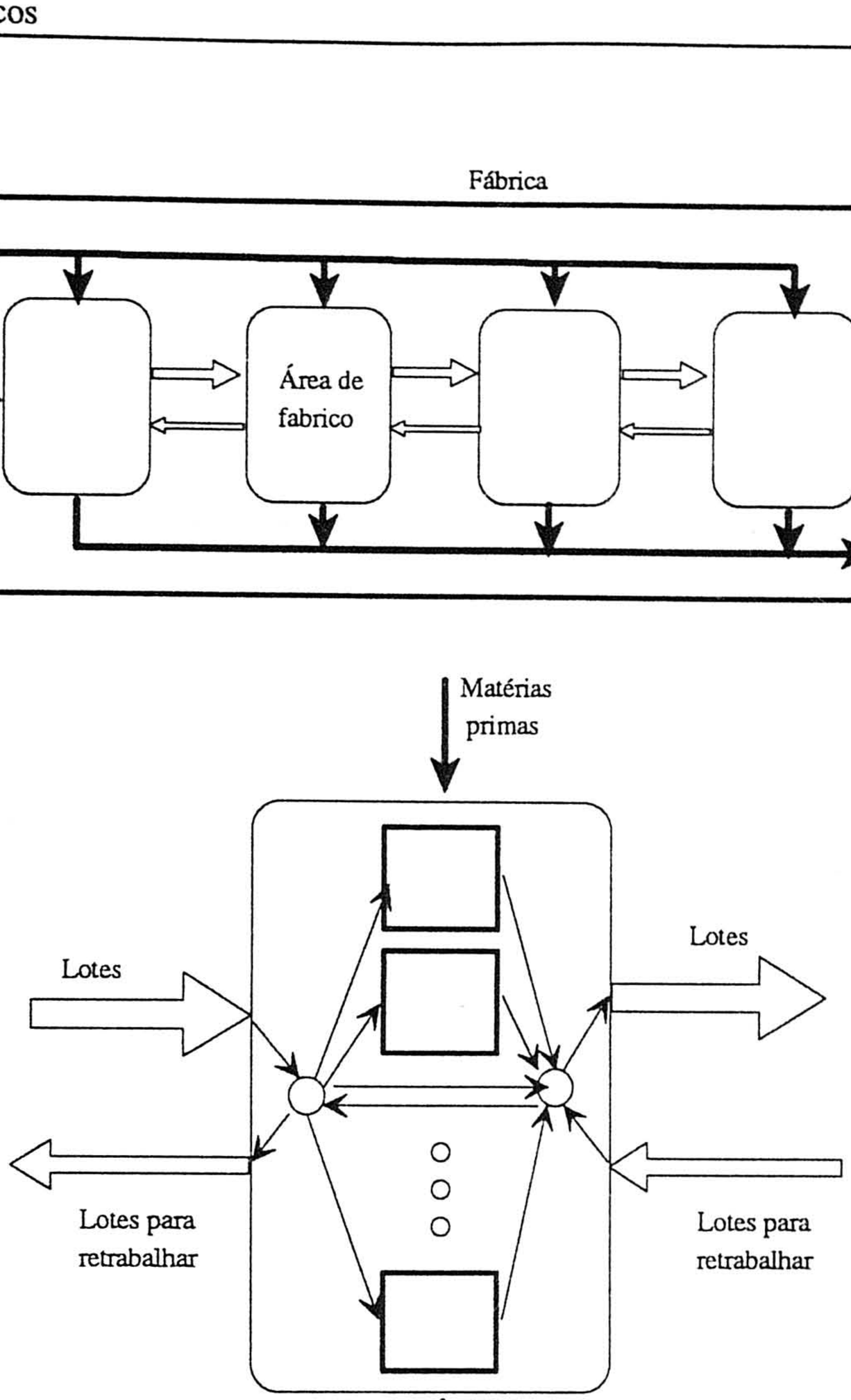
Cada área é constituída por equipamentos que executam uma função semelhante. Por exemplo uma área de moldagem é constituída por prensas de molde e equipamentos auxiliares, como equipamentos de transporte, de aquecimento do material, etc.

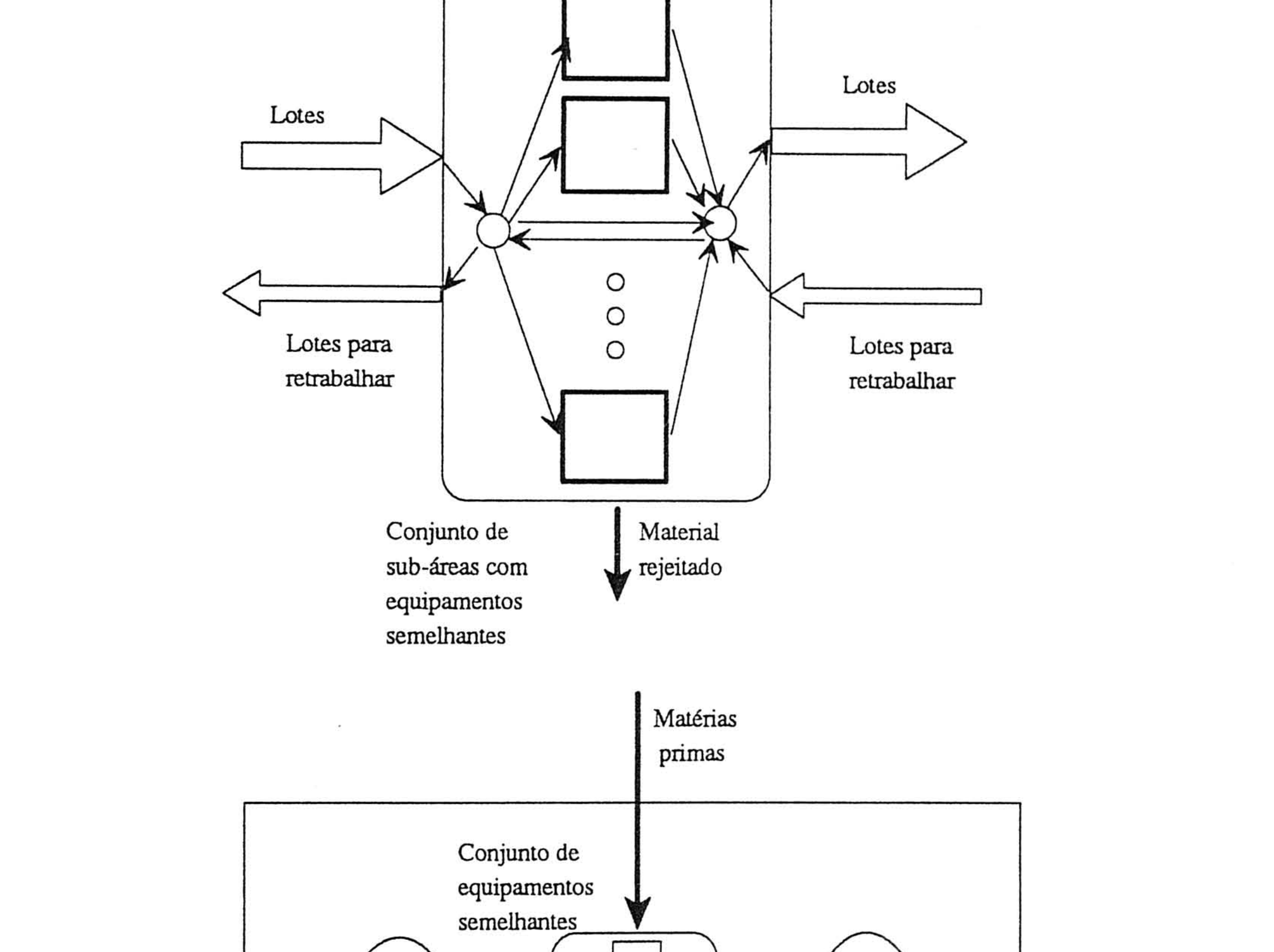
Entre cada área, existe habitualmente uma zona para armazenar produtos em curso de fabrico. Estes produtos já foram processados na área anterior e estão à espera da disponibilidade de equipamento na área seguinte para serem processados. Esta zona é denominada de zona de inventário em curso de fabrico. Este inventário serve como um *buffer* de segurança relativamente a problemas de produção na(s) área(s) anterior(es). Se existir uma paragem anormal numa dada área, as áreas seguintes podem continuar a laborar durante algum tempo, diminuindo em consequência o inventário que existe após a área com problemas. Quanto maior for este inventário, mais tempo tem uma área para se recuperar de uma situação anormal sem afectar as áreas seguintes.

No entanto, o somatório de todos os inventários de produtos em curso de fabrico, representa um custo elevado para a empresa. São produtos que contêm matéria prima e mão de obra incorporada em grau variável e que se mantêm permanentemente na empresa (não os produtos específicos, mas a sua quantidade). A gestão destes inventários tem sido objecto de grande investigação nos últimos anos de forma a que se mantenham num nível mínimo. mas permitindo uma gestão ainda flexível dos equipamentos.

O uso de técnicas de simulação para o cálculo dos equipamentos necessários, como é o caso deste trabalho, traz um grande contributo para uma boa gestão destes recursos. Isto é obtido quer pela experimentação rápida de diferentes alternativas, quer pelo cálculo mais eficiente da quantidade de cada tipo de equipamento necessário para a produção a realizar.


Matérias primas
Fábrica
Encomendas
Área de fabrico
Produtos acabados
Material rejeitado


Matérias primas
Lotes
Lotes
Lotes para retrabalhar
Lotes para retrabalhar
Conjunto de sub-áreas com equipamentos semelhantes
Material rejeitado


Matérias primas
Conjunto de equipamentos semelhantes
Inventário à espera de processo
Inventário à espera de qualificação
Lotes
Lotes
Material rejeitado

Existem essencialmente dois tipos de produção: a produção contínua e a produção por lotes.

A produção contínua consiste na existência de uma linha de fabrico que está permanentemente a processar um dado produto. Habitualmente, neste caso, cada produto diferente é produzido numa linha de fabrico diferente. Um caso típico de produção contínua é o fabrico de bebidas engarrafadas.

A produção por lotes consiste no processamento agrupado de um conjunto de produtos de características semelhantes. Por exemplo, ao fabricar um lote de 20 sofás de um dado tipo, os sofás apenas passam para a área de fabrico seguinte quando todos os 20 tiverem completado o processo na área anterior. Este tipo de fabricação embora muito mais complexo de gerir permite a obtenção de maior eficiência económica, melhor controlo de qualidade e a satisfação mais rápida de uma dada encomenda.

É neste segundo caso, produção por lotes, que se aplica o trabalho realizado. A produção contínua não necessita de sistemas complexos de gestão devido à sua simplicidade.

Em termos de gestão de inventários de produtos acabados existem dois tipos de produção: produção para armazenagem e produção por encomenda.

A produção para armazenagem consiste no fabrico constante de produtos. Este é ainda o tipo de produção mais usado, embora a sua aplicação tenda a diminuir. O departamento de vendas vai recebendo as várias encomendas, mandando os produtos do armazém para o cliente. A linha de produção tem um regime de trabalho essencialmente constante. Nas alturas de menor procura o nível armazenado sobe, para descer nas alturas de maior procura.

Este tipo de produção tem o mesmo problema que o apresentado pelos inventários em curso de fabrico. O inventário de produtos acabados constitui um custo enorme para a empresa. É dinheiro que está parado fechado dentro de um armazém. Também apresenta vários problemas em termos de qualidade. Um problema no produto introduzido por um dado equipamento pode só ser detectado vários meses depois quando o produto for utilizado. Por este motivo cada vez é maior o número de empresas que tenta passar para um sistema de produção por encomendas.

A produção por encomendas consiste em produzir um dado produto só após a recepção de uma encomenda por parte de um cliente. A encomenda do cliente dá origem a um (ou possivelmente mais, caso seja elevada) lote. Para a produção deste tipo ser eficiente é necessário existir um grande conhecimento e domínio das técnicas de produção necessárias ao fabrico dos produtos. É necessário a redução significativa dos tempos de fabrico, de forma a que um cliente não necessite de esperar muito tempo por uma encomenda realizada.

Este tipo de produção é ainda mais eficiente quando se trata de fabricantes de componentes, e não de produtos para o consumidor final. Neste caso, é conveniente a colaboração entre fornecedor-cliente para um conhecimento antecipado (embora sem compromisso) das necessidades futuras. Uma boa gestão de recursos permite obter custos mais baixos, que se podem traduzir no cliente por um preço mais reduzido dos produtos a adquirir. Assim todos saiem a ganhar duma estratégia deste tipo. Da colaboração referida sai um plano de produção (ou encomendas) para um dado período de tempo futuro.

O trabalho realizado insere-se neste tipo de ambiente industrial. Do conhecimento do plano de encomendas futuro, é feita a simulação da situação instalada, verificando-se se existe falta ou excesso de equipamentos para o período de tempo considerado. Caso exista excesso, pode ser considerada a sua venda, a sua utilização por outras fábricas semelhantes da mesma companhia em localização diferente, ou a utilização desse período para uma manutenção mais pormenorizada dos equipamentos. No caso de falta, deve ser considerada a encomenda de mais equipamento na altura apropriada. A aquisição de equipamento industrial é habitualmente demorada, pelo que uma boa previsão é importante para uma boa gestão futura.

II.4.2 Equipamentos de produção

Conforme referido anteriormente cada área de fabrico é constituída por equipamentos que executam uma dada função. Podem existir equipamentos diferentes para processar produtos diferentes assim como podem existir equipamentos auxiliares, mas a função final a realizar é sempre semelhante.

No trabalho realizado não me preocupo com os sistemas auxiliares pois os mesmos são facilmente calculados a partir do conhecimento da quantidade de equipamento base necessário.

Para efeitos da criação de um modelo fabril é necessário considerar um modelo de um equipamento de produção, suficientemente simples para se poderem tirar resultados do modelo em tempo útil, mas suficientemente completo para poder representar de forma fiável o equipamento em causa.

No trabalho refiro algumas variáveis que passo a descrever:

Yield : rendimento da operação. Quando um lote é introduzido num dado equipamento é normal que o número final de unidades seja inferior ao número de unidades inicial. Isto acontece porque algumas unidades têm problemas durante o processo devendo por isso ser retiradas. O rendimento final do lote será a multiplicação dos rendimentos obtidos em todas as áreas por que passou. O rendimento médio de cada área é conhecido para cada tipo de produto. Como cada lote é resultado de uma encomenda, é necessário que o lote inicie o processo de fabrico com uma quantidade que tenha em conta o rendimento total de fabrico desse tipo de lote mais uma pequena margem de segurança.

PPH : *parts per hour* . É um índice que nos dá a velocidade de trabalho de um dado equipamento. Corresponde ao número de unidades que em média esse equipamento consegue processar numa hora de trabalho.

Downtime : período de tempo que um dado equipamento está parado quer devido a avaria, quer devido a manutenção.

fluxo: percurso que um lote deve percorrer durante o processo de produção. O fluxo depende do tipo de produto, e indica a sequência de equipamentos necessários para produzir esse tipo de material.

MTBA : tempo médio entre assistências do operador da máquina (*Mean Time between Assists*).

MTBF : tempo médio entre avarias (*Mean Time Between Failures*).

II.4.3 Fabrico de circuitos integrados

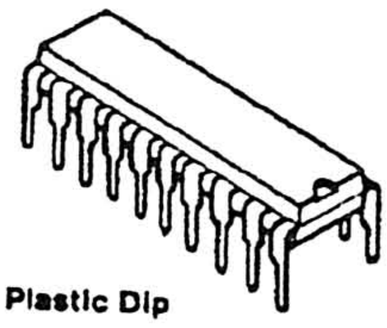
II.4.3.1 Introdução

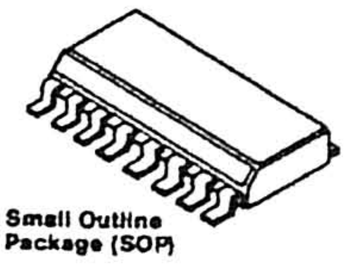
O fabrico de circuitos integrados em particular e o dos semicondutores em geral, é caracterizado pela existência de um grau de complexidade elevado. Quer o domínio de aplicação está claramente no limiar da tecnologia, quer as formas de planeamento e organização necessitam da aplicação de técnicas evoluídas.

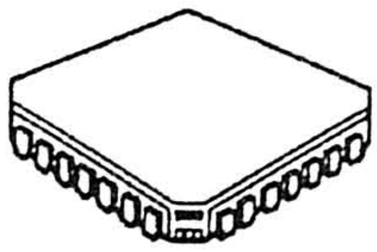
No mercado de circuitos integrados, tanto os custos à entrada (custos associados à entrada no mercado de um novo fabricante), como os custos à saída (idem, mas à saída), deste tipo de mercado são muito elevados. Assim, apenas empresas possuidoras de um conhecimento tecnológico elevado e detentoras de forte capacidade financeira se introduzem neste tipo de mercados. Isto leva à existência de forte concorrência. Neste tipo de ambiente as preocupações com a qualidade dos produtos fabricados e os custos dos mesmos são elevadas.

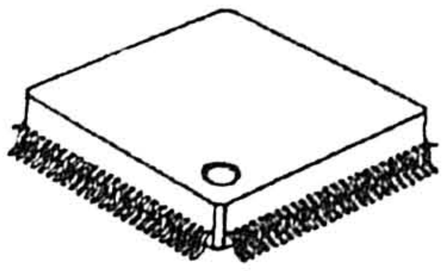
O cliente típico de um fabricante de circuitos integrados são os fabricantes de produtos electrónicos, como por exemplo computadores, televisores, rádios, máquinas de lavar, etc.

Um circuito integrado é um componente electrónico de elevado nível de integração. Em termos de componentes básicos, por exemplo transistores, um circuito integrado contém sempre alguns milhares de transistores podendo o seu número ultrapassar o milhão.


Plastic Dip


Small Outline Package (SOP)


Plastic Leaded Chip Carrier (PLCC)


Square Quad Flat Pac (SQFP)

A comunicação eléctrica com o exterior é realizada através de pinos ou pernos metálicos. Existem vários tipos de circuitos integrados que são resultado da disposição e quantidade desses mesmos pinos e do tipo de envólucro exterior. Um determinado *device* (um circuito com determinadas características) pode ser

fabricado em vários tipos de formato dependendo do interesse manifestado pelos clientes.

Existem muitos tipos de circuitos que são produzidos desta forma. Alguns exemplos são microprocessadores, memórias, PALs (*Programmable Array Logic*), e GPL (*General Purpose Logic*).

A descrição de produção que será feita refere-se a GPL embora os outros tipos de circuitos tenham uma produção muito semelhante. Como exemplos de formato externo serão usados os formatos DIP (*Dual In line Package*) e SOIC (*Small Outline Integrated Circuit*).

II.4.3.2 Fabrico de circuitos integrados

A produção de um circuito integrado divide-se essencialmente em três áreas de responsabilidade, nem sempre na mesma localização.

A primeira área é a responsável pela promoção da empresa e dos seus produtos, obtendo dos clientes encomendas e fazendo chegar ao cliente os produtos pedidos nas condições e períodos que o mesmo pretende. Trata resumidamente da função comercial da empresa. Quando um cliente faz uma encomenda para ser recebida num dado período, esta área trata do planeamento necessário para que tal aconteça. Usando o conhecimento do tempo necessário para fabricar um determinado circuito, executa no momento apropriado uma encomenda interna à área seguinte de um lote constituído pela quantidade apropriada de um determinado circuito integrado pretendido pelo cliente.

A segunda área, por vezes denominada PCC (*Product-Customer Center*), trata do fabrico do circuito integrado em si, sem preocupações com a forma envolvente final. Trata do projecto de novos circuitos integrados, do desenho, da interface técnica com o cliente, da produção das várias máscaras necessárias para a criação dos vários níveis existentes, da purificação do material semicondutor necessário, do fabrico dos circuitos em quantidade, e finalmente de um primeiro teste funcional. Na fase final do fabrico nesta área, os circuitos integrados (*dies*) encontram-se agrupados (num número que vai desde algumas centenas a vários milhares) em *wafers*. As várias *wafers* que constituem o lote são colocadas em caixas apropriadas e enviadas para a terceira área.

A terceira área, por vezes denominada FCC (*Fabrication Customer Center*), trata da individualização dos vários circuitos e da sua instalação num meio adequado para a sua utilização futura. Aqui o lote recebido será processado e finalmente enviado para um armazém controlado pela primeira área referida, donde os lotes seguirão para os clientes na altura apropriada.

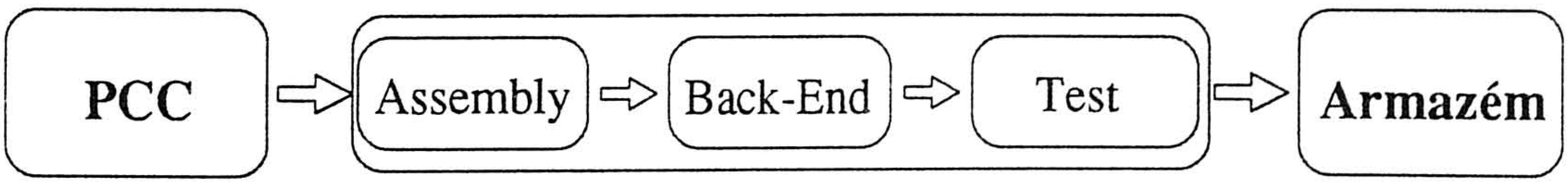
O caso prático tratado neste trabalho desenvolve-se nesta terceira área. Para uma melhor compreensão do trabalho vou aqui apresentar a estrutura da mesma. Uma pequena descrição do processo de fabrico existe no anexo C, onde poderá ser consultada.

II.4.3.3 Descrição de um FCC

A descrição seguinte respeita apenas ao sector de produção e ignora algumas pequenas áreas sem interesse para o estudo em causa.

A partir daqui sempre que me referir ao fabrico de um circuito integrado deve-se entender a parte do fabrico que é realizada num FCC.

Um FCC é constituído por três grandes áreas por sua vez divididas em mais áreas. Dependendo do grau de integração e do tipo de circuito a produzir podem ou não existir diversas áreas de fabrico. O caso a descrever é de média integração e nem todos os tipos de circuito passam no seu percurso de fabrico por todas as áreas. Essas três grandes áreas são denominadas *Assembly*, *Back-end* e *Test*.



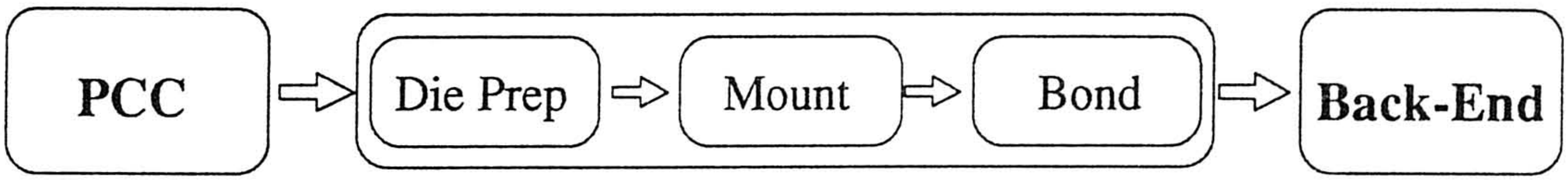
O processamento de circuitos integrados é habitualmente feito por lotes, ou seja, por conjuntos de unidades com características semelhantes. Não é habitual a produção em contínuo, ou seja, o fabrico permanente de um produto com um determinado conjunto de características. A produção por lotes pode se necessário

funcionar de forma semelhante à produção contínua, pela divisão dessa produção em pequenos lotes.

A produção por lotes apresenta diversas vantagens, especialmente quando o número de tipos de produtos a fabricar é elevado e os equipamentos são flexíveis. Não existe a necessidade de se criar uma linha de produção para cada tipo de produto. Além disso não se fabricam produtos para serem armazenados, sendo apenas produzidos lotes que correspondem a encomendas reais. Se pretendessemos ter fabrico para armazenagem, este poderia ser introduzido através da execução de encomendas internas (feitas pelo armazém final) para além das encomendas externas (realizadas pelos clientes). Para este sistema funcionar bem é necessário recorrer a sofisticadas técnicas de gestão de produção, a mais importante das quais é habitualmente conhecida por produção *Just-In-Time*.

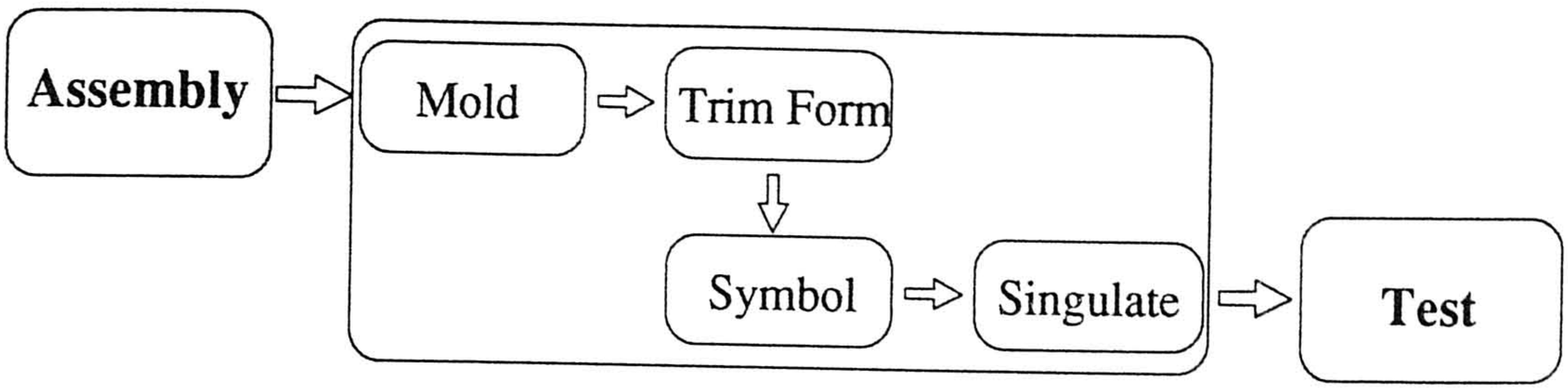
II.4.3.3.1 *Assembly*

O *assembly* é constituído por três subáreas. Essas áreas são denominadas *Die-prep*, *Mount* e *Bond*. E a área responsável pelo fabrico inicial do circuito. Todas estas áreas têm de se realizar dentro de uma "sala limpa", ou seja uma (ou várias) sala(s) em que o tamanho das partículas que circulam no ar é muito reduzido e permanentemente controlado.



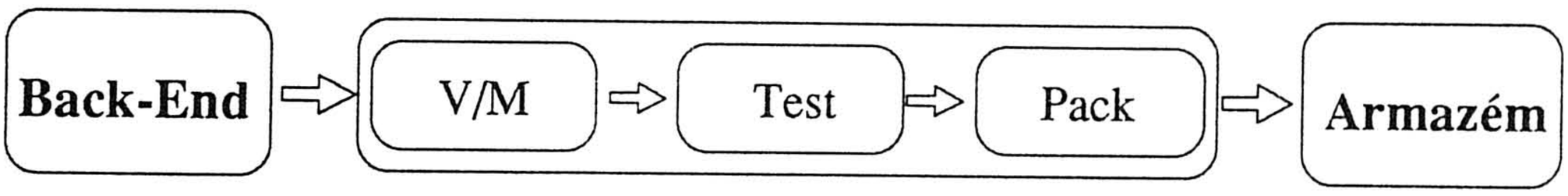
II.4.3.3.2 *Back-end*

O *back-end* é constituído por quatro subáreas. Essas áreas são denominadas *Mold*, *Trim-Form*, *Symbol* e *Singulate*. É a área responsável pelo fabrico final do circuito.

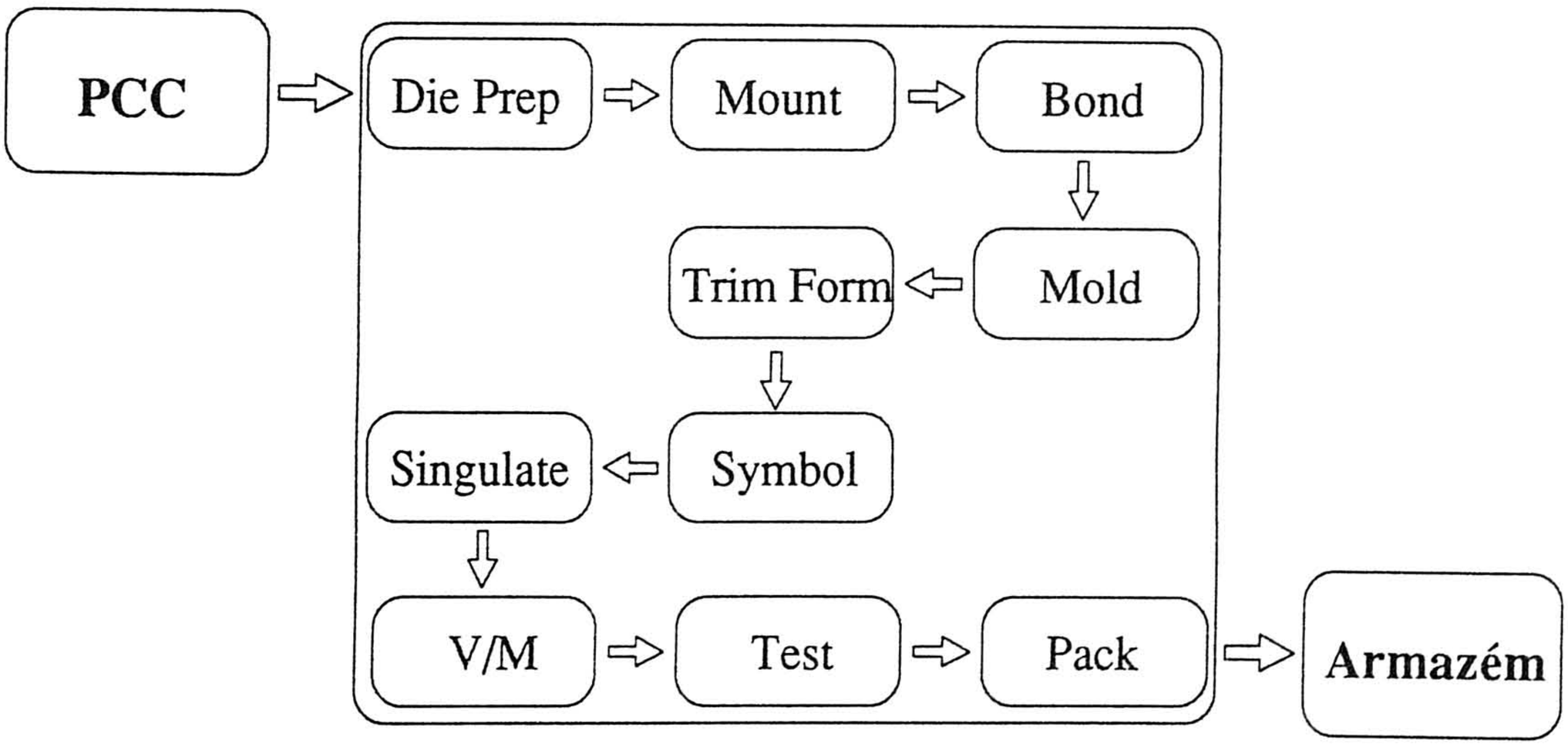


II.4.3.3.3 *Test*

O *test* é constituído por três subáreas. Essas áreas são denominadas *V/M*, *Test* propriamente dito e *Pack*. É a área responsável pelo teste final do circuito. As preocupações com a qualidade, embora sempre existentes ao longo de todo o processo de fabrico, tomam aqui um relevo especial, pois é o último filtro existente antes das unidades serem entregues ao cliente.



Apresenta-se de seguida um resumo do processo de fabrico total:



II.5 Estatística

II.5.1 Noções básicas de simulação-técnica

Um número significativo de modelos de simulação utiliza variáveis aleatórias (taxa de chegadas de lotes a um equipamento, duração do serviço prestado).

Nesses casos o resultado do modelo não será um valor preciso, mas apenas um estimador(es) de variáveis aleatórias. Daí que, um aspecto importante de um modelo de simulação seja a análise estatística.

Antes de mais há o problema de geração de variáveis aleatórias, com amostragem a partir de distribuições apropriadas. Depois põe-se o problema de saber se o modelo representa (em termos estatísticos) a situação real. Finalmente, é preciso saber durante quanto tempo deve "correr" o modelo para que os resultados sejam estáveis e significativos do ponto de vista estatístico.

O problema de geração de amostras aleatórias pode ser resolvido em duas fases:

(i) Geração de valores distribuídos uniformemente;
(ii) Transformação de uma variável uniforme noutra com qualquer distribuição.

II.5.2 Números aleatórios

Números aleatórios são números uniformemente distribuídos no intervalo [0,1].

As propriedades desejáveis de uma sequência de números aleatórios são:

(i) Independência
(ii) Distribuídos uniformemente
(iii) Reproduzível
(iv) Sem ciclos para qualquer duração
(v) Passíveis de serem gerados a alta velocidade
(vi) Pouco exigentes em memória necessária

Uma alternativa seria armazenar tabelas de números aleatórios em disco, mas é mais rápido e simples usar uma relação de recorrência. No entanto os números gerados por este processo não são de facto "aleatórios" e por isso são normalmente designados por pseudo-aleatórios.

Os métodos mais usados para a geração de números pseudo-aleatórios são os métodos congruenciais. Em relação a estes (que normalmente se encontram disponíveis na maioria dos computadores) apenas há que garantir as duas primeiras propriedades referidas acima.

A classe Random do Smalltalk será a responsável pela obtenção dos valores aleatórios a usar neste trabalho.

Esta classe usa um gerador do tipo congruencial linear baseado numa recomendação de "Numerical Recipes" (W.H. Press e outros, Cambridge University Press, 1986, pp. 191-199).

II.5.3 Simulação de variáveis aleatórias

Seja x uma variável aleatória com função densidade de probabilidade

$$f(x)\,dx = P(x <= X <= x+dx)$$

e função distribuição de probabilidade

$$F(x) = P(X <= x) = \int_{-\infty}^{x} f(x)\,dx$$

se y for uma variável aleatória uniforme estandardizada no intervalo [0,1], então para uma observação de y daquela variável

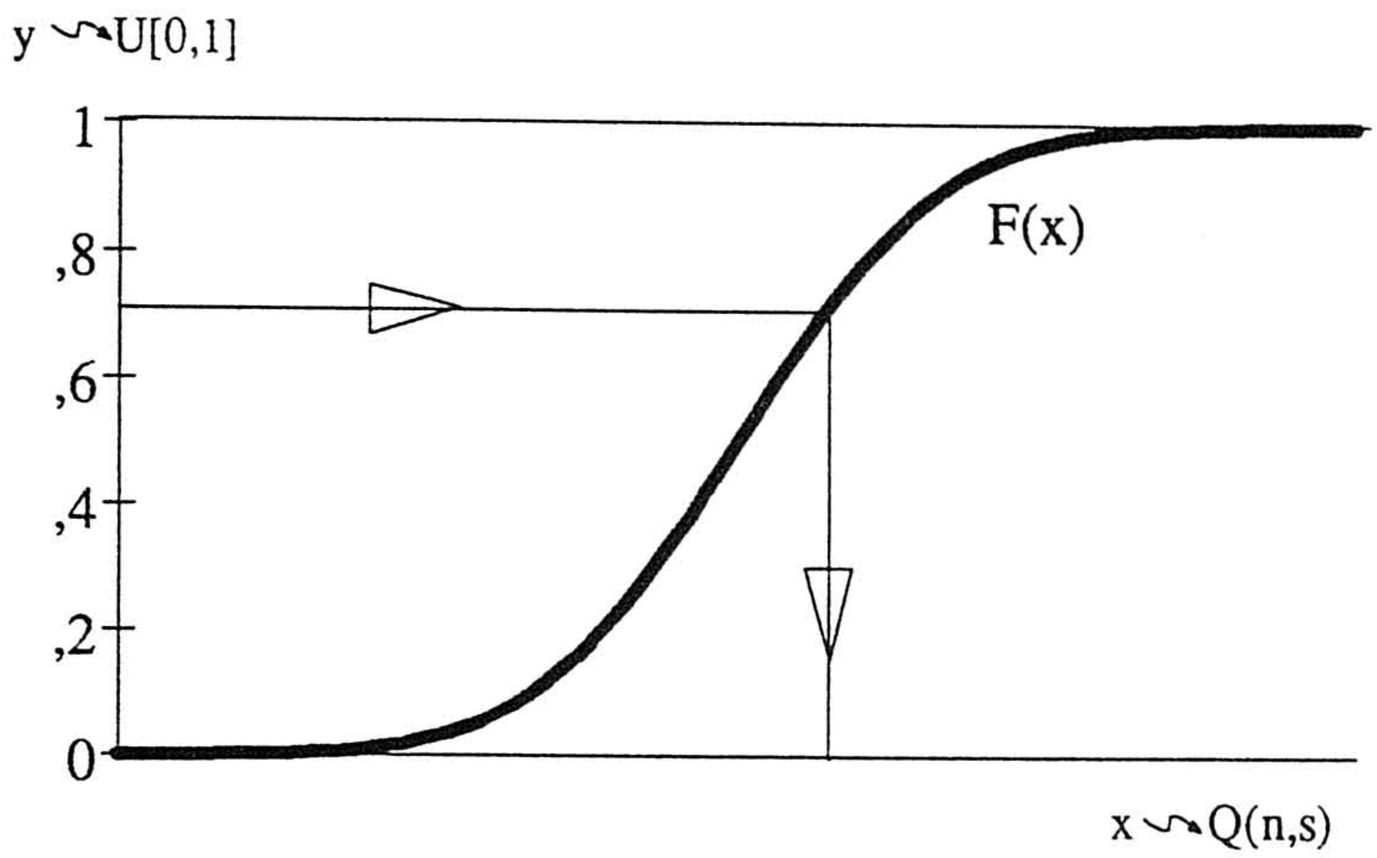
$$P(y <= F(x)) = F(x)$$

$$P(F^{-1}(y) <= x) = F(x)$$

porque F(x) é monótona não decrescente. Por isso

$$F^{-1}(y) = F(x)$$

o que é equivalente a dizer que $F^{-1}(y)$ tem a mesma distribuição de F (x).



II.5.4 Distribuições de probabilidade

Para utilização nesta simulação assim como em outras, foram criadas várias classes que permitem obter vários tipos de distribuição, do tipo discreto e do tipo contínuo. As distribuições que foram definidas são:

- Distribuições discretas:
  - Bernoulli;
  - Binomial;
  - Geométrica;
  - Poisson;
- Distribuições contínuas:
  - Uniforme;
  - Exponencial;
  - Gamma;
  - Normal.

A forma como estas distribuições foram implementadas facilita a introdução futura de novas distribuições que sejam consideradas necessárias.

A implementação das classes que vão permitir a existência destas distribuições será comentada no próximo capítulo. A descrição teórica destas distribuições existe no anexo B para consulta. Neste anexo são dados exemplos de aplicações das diferentes distribuições ao caso da simulação de ambientes industriais.

Um estudo mais aprofundado deste tema pode ser realizado em [B1], [B2], [B3] ou [B4].

III. Simulador de ambientes industriais - SimAI

III.1 Introdução

Uma simulação é a representação de um sistema de objectos no mundo real ou fantasia. O objectivo de criar uma simulação por computador é o de proporcionar um ambiente de trabalho que permita compreender melhor a situação que pretendemos simular. Compreender por exemplo, o comportamento ou evolução do inventário em processo (um tipo de fila de espera), a quantidade de trabalho dos operadores, o tempo total de fabrico de um lote, etc.

Certos tipos de simulação consistem essencialmente numa relação tipo servidor-cliente. Representam situações em que temos servidores (por exemplo, equipamentos) em determinados lugares onde são abordados por clientes (por exemplo, lotes) que pretendem obter um determinado serviço. Se um dado servidor não está disponível, o cliente deverá ficar numa fila de espera (o chamado inventário em processo) até poder ser atendido. Quando um servidor fica disponível, o cliente que está à espera há mais tempo (ou qualquer outra regra a definir) é retirado da fila de espera e é atendido.

Habitualmente, uma dada simulação tem vários tipos de servidores e de clientes, cada um dos quais com uma agenda de serviços a realizar. O caso dos ambientes de fabrico industrial é um exemplo deste tipo de situações.

Uma simulação por computador torna possível a obtenção de estatísticas acerca de diferentes situações de forma a testar novas ideias em termos de organização, assim como permite o cálculo dos recursos necessários para responder a uma dada procura conhecida.

Os objectos que participam numa simulação operam de forma essencialmente independente uns dos outros. Por isso, é necessário criar os meios de coordenação e sincronização das actividades dos vários objectos da simulação.

Habitualmente a coordenação é feita através do mecanismo de passagem de mensagens. No entanto, alguns objectos devem sincronizar as suas acções em

momentos críticos, por exemplo, alguns objectos só poderão exercer as suas funções se um dado recurso dinâmico estiver disponível em determinado momento.

As classes Process, Semaphore, e SharedQueue, existentes no Smalltalk-80, proporcionam os meios de garantir a referida sincronização.

Para suportar uma descrição de ambientes de simulação foram introduzidos mecanismos de forma a coordenar:

- o uso de recursos de tamanho fixo;
- o uso de recursos flutuantes;
- a aparência de simultaneidade das acções de diferentes objectos.

Os recursos de tamanho fixo podem ser de dois tipos: consumíveis e não consumíveis. Por exemplo, plástico para moldagem é um recurso fixo e consumível da área de molde, enquanto equipamentos são um recurso fixo e não consumível.

Recursos flutuantes são habitualmente referidos como renováveis ou de sincronização produtor/consumidor. A área de molde pode modelizar o fornecimento de plástico como um recurso flutuante porque o seu aprovisionamento pode ser renovado. Também existem recursos que são ao mesmo tempo renováveis e não consumíveis. Este tipo de recursos pode ser imaginado por exemplo na simulação de uma agência de aluguer de automóveis: os carros são recursos renováveis pois carros novos são fabricados e comprados pela agência para serem alugados, e são também recursos não consumíveis pois depois de alugado o carro é devolvido para utilização de outros clientes.

Na realidade, grande parte dos recursos definidos como não consumíveis são consumíveis. Os equipamentos de fabrico ou os carros chegarão a um ponto em que já não estão em condições de continuarem a funcionar. Entende-se assim a definição de não consumível como referente ao período temporal de interesse para o âmbito da simulação executada.

Quando a acção de dois objectos da simulação necessita de ser sincronizada para dar a aparência de estarem a executar uma tarefa em conjunto, os dois objectos são referidos como estando numa relação servidor/cliente. Por exemplo um equipamento de teste necessita de ter um lote para proceder ao teste das unidades. Diz-se que o servidor é um recurso coordenado, ou seja é um objecto da simulação cujas tarefas apenas podem ser executadas na presença de um ou mais clientes.

Um aspecto importante das simulações é que elas modelizam situações que variam ao longo do tempo. Essas actividades são habitualmente relacionadas em termos de tempo; em certos momentos ou durante certos intervalos os acontecimentos tomam lugar. Logo, todas estas acções têm de ser sincronizadas em termos de tempo. Muitas vezes, esta noção de tempo é ela própria simulada.

Existem várias formas de representar as acções dos objectos da simulação relativamente ao tempo real ou simulado.

Numa das aproximações o relógio funciona de uma forma normal. A cada incremento do relógio, é dada oportunidade a todos os objectos de executarem as suas funções. O relógio age como um agente de sincronização da simulação, proporcionando a oportunidade da aparência de paralelismo pois o relógio espera até todas as acções a executar naquele momento estarem terminadas. É comum nenhuma acção ser executada a um dado avanço do relógio.

Noutra alternativa, o relógio será avançado para o momento em que a próxima acção deverá ter lugar. A implementação de uma simulação usando esta aproximação necessita da manutenção de uma fila de espera de acontecimentos ordenados por ordem de tempo de simulação. De cada vez que um acontecimento é executado, o próximo é retirado da fila de espera e o relógio é avançado para o momento apropriado da execução.

O sistema SimAI usa esta aproximação, que é vulgarmente conhecida como simulação orientada por acontecimentos.

Neste capítulo serão descritas as classes mais importantes para a implementação do sistema desenvolvido. Começou-se por criar uma estrutura que permitisse a modelização de simulações orientadas por acontecimentos e correspondente recolha de estatísticas. Esta estrutura baseou-se na recomendada por Goldberg e Robson ([B12]) para uma versão anterior do Smalltalk, tendo-se juntado as classes necessárias para a implementação final do sistema SimAI.

Vou começar por descrever as classes que foram desenvolvidas para a obtenção de distribuições de probabilidade, que são fundamentais em simulação.

Depois descreverei a estrutura de classes em que a simulação será implementada. A classe SimulationObject descreve a estrutura genérica dos objectos da simulação. Uma instância da classe Simulation mantém o relógio da simulação e a fila de espera dos acontecimentos.

De seguida serão descritas as classes responsáveis pela recolha de dados relacionados com a simulação.

Será então descrita a utilização e coordenação de diversos tipos de recursos.

Finalmente entrarei em alguns pormenores do sistema SimAI.

```
Object ()
    Stream ()
        ProbabilityDistribution ()
            DiscreteProbability ()
                Bernoulli ('prob')
                    Binomial ('N')
                    Geometric ()
                Poisson ('mu')
                SampleSpace ('data')
                    SampleSpaceWithReplacement ('rand')
                        SampleSpaceWithoutReplacement ()
            ContinuousProbability ()
                Exponencial ('mu')
                    Gamma ('N')
                Normal ('mu' 'sigma')
                Uniform ('startNumber' 'stopNumber')
Simulation ('resources' 'currentTime' 'eventQueue' 'processCount')
    StatisticsWithSimulation ('statistics')
SimulationObject ()
    EventMonitor ('label')
DelayedEvent ('resumptionSemaphore' 'resumptionCondition')
    WaitingSimulationObject ('amount' 'resource')
Histogram ('tallyArray' 'lowerBound' 'upperBound' 'step' 'minValue'
        'totalValues' 'extraEntries')
SimulationObjectRecord ('entranceTime' 'duration')
Resource ('pending' 'resourceName')
    ResourceCoordinator ('whoIsWaiting')
    ResourceProvider ('amountAvailable')
```

Estrutura base das classes de simulação

III.2 Distribuições de probabilidade

A classe ProbabilityDistribution foi criada como subclasse da classe Stream de forma a usufruir dos métodos aí existentes.

Esta classe, que é base de todas as outras classes de distribuição de probabilidade cria o protocolo para obter uma ou mais amostras da distribuição, e para calcular as funções densidade de probabilidade e distribuição cumulativa.

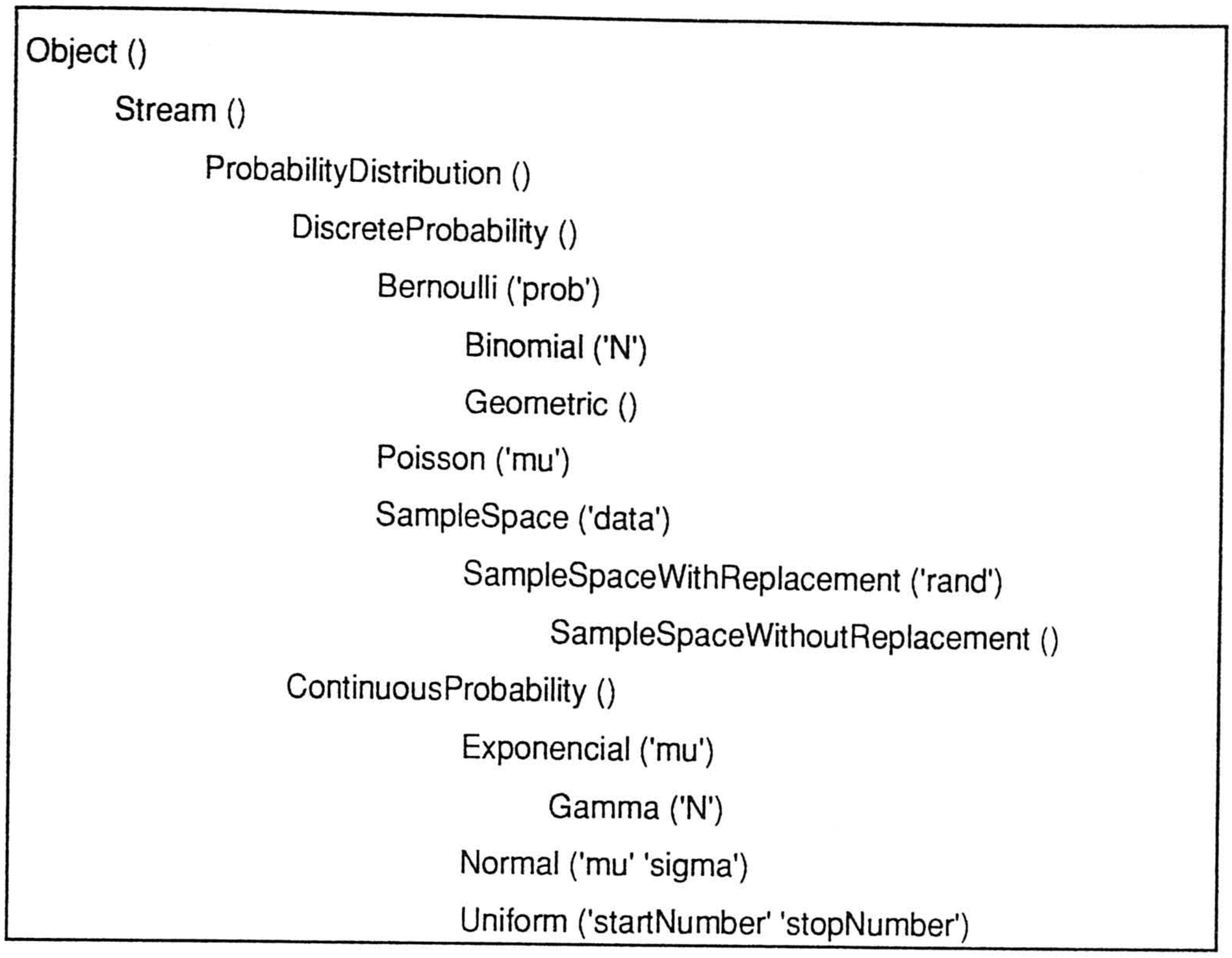
Tal como a classe Random, a classe ProbabilityDistribution é uma sequência que acede aos seus elementos gerando-os de forma algorítmica. Quando é necessária uma amostra, a mensagem next é enviada à distribuição. Esta classe implementa o método next retornando o valor resultante da mensagem inverseDistribution: var, em que o argumento var é um número aleatório entre 0 e 1. As subclasses de ProbabilityDistribution implementarão esta mensagem de forma a mapear o intervalo [0,1] no seu próprio espaço de amostragem. A mensagem next: é herdada da classe Stream.

Antes de se usar a classe definida será necessário inicializar a variável U através da mensagem: ProbabilityDistribution initialize.

Devido à utilidade para várias das subclasses foi aqui também implementado um método que permite calcular o número de maneiras de que se pode obter uma amostra de tamanho *m* de um conjunto de tamanho *n* sem reposição.

Os dois tipos de distribuição de probabilidade, discreta e contínua, foram definidos como subclasses de ProbabilityDistribution, implementando cada um deles a função distribuição cumulativa que por sua vez depende da função densidade de probabilidade.

As subclasses de DiscreteProbability e ContinuousProbability implementam as funções densidade de probabilidade e distribuição inversa.



Estrutura de classes responsável pelas distribuições de probabilidade

III.2.1 Distribuições discretas

Conforme referido, a classe DiscreteProbability proporciona a base para a definição das distribuições de probabilidade discretas. Foram definidas quatro tipos de distribuições: Bernoulli, Binomial, Geométrica e Poisson.

Para auxílio das outras classes foi definida ainda uma classe denominada SampleSpace como subclasse de DiscreteProbability.

Esta classe implementa o conceito habitual de que a probabilidade de um acontecimento é a razão entre os casos prováveis e os casos possíveis. A probabilidade de ocorrência de cada caso é igual, pelo que essa probabilidade corresponde ao inverso do tamanho do conjunto. A função densidade tem apenas de somar essa probabilidade tantas vezes quantos forem os elementos do conjunto, de que pretendemos obter a densidade de probabilidade. A implementação da função de distribuição cumulativa é herdada da superclasse.

III.2.1.1 Distribuição de Bernoulli

De acordo com a especificação da classe Bernoulli podemos criar uma distribuição de Bernoulli usando uma expressão do tipo Bernoulli parameter: 0.17. Neste caso teríamos criado uma distribuição de Bernoulli com uma probabilidade de sucesso igual a 0,17. Na distribuição de Bernoulli a média é igual à probabilidade de sucesso, pelo que os dois termos são por vezes usados indistintamente.

A classe contém uma variável de instância prob que corresponde à probabilidade de um dos dois possíveis acontecimentos. Para evitar confusões, esta probabilidade será a do sucesso. Este número será portanto um valor entre 0.0 e 1.0. A função densidade de probabilidade faz corresponder os dois acontecimentos possíveis, 1 ou 0, com a probabilidade prob ou o seu inverso. A função de distribuição cumulativa é herdada da superclasse e apenas retorna os valores prob ou 1.

III.2.1.2 Distribuição Binomial

A distribuição binomial representa a repetição de N tentativas de Bernoulli independentes. O seu resultado é o mesmo da distribuição de Bernoulli para N = 1. Devido à correspondência entre estas duas distribuições a classe Binomial foi criada como uma subclasse da classe Bernoulli. Ela contém uma variável de instância N que corresponde ao número de tentativas.

A implementação da classe Binomial não usa directamente a expressão da definição da distribuição, mas uma sua transformação. A expressão pode ser vista como o número de formas de obter x sucessos de N tentativas, dividido pelo número de formas de obter x sucessos em x tentativas. A implementação usa o método computeSample: a outOf: b herdado da superclasse ProbabilityDistribution.

III.2.1.3 Distribuição Geométrica

Vamos supor, como exemplo que em média chegam a uma dada máquina 2 lotes por hora. Esta informação estatística pode ser expressa como:

```
amostra := Geometric mean: 24/2
```

A função densidade de probabilidade pode ser usada para calcular a probabilidade de acontecerem *N* tentativas antes do próximo sucesso. Por exemplo a probabilidade de se passarem 30 minutos antes do próximo lote chegar pode ser dada por:

```
amostra density: 0.5
```

A função de distribuição cumulativa pode ser usada para calcular se chegará algum lote no período compreendido entre 2 a 3 horas a partir de agora:

```
amostra distribution: (2 to: 3)
```

III.2.1.4 Distribuição de Poisson

A resposta à mensagem next dá o número de acontecimentos que ocorrerão na próxima unidade de tempo. A função densidade de probabilidade de x determina a probabilidade de na próxima unidade de tempo, x acontecimentos se realizarem. A função de distribuição cumulativa de x determina a probabilidade de numa unidade de tempo ocorrerem x ou menos acontecimentos.

III.2.2 Distribuições contínuas

Conforme referido, a classe ContinuousProbability proporciona a base para a definição das distribuições de probabilidade contínuas. Foram definidas quatro tipos de distribuições: Uniforme, Exponencial, Gamma e Normal.

III.2.2.1 Distribuição Uniforme

A classe Uniform extende as capacidades da classe Random do sistema gerando uma variável aleatória dentro de um dado intervalo como resposta à mensagem next.

III.2.2.2 Distribuição Exponencial

A classe Exponential implementa esta distribuição.

III.2.2.3 Distribuição Gamma

A classe Gamma foi definida como subclasse da classe Exponential. Cada instância desta classe representa o N-ésimo acontecimento e a probabilidade de ocorrência desse acontecimento (herdada da superclasse Exponential).

III.2.2.4 Distribuição Normal ou de Gauss

A classe Normal implementa esta distribuição.

III.3 Estrutura base de simulação

III.3.1 Os objectos de uma simulação

Considere-se o caso de uma fábrica. Alguns dos componentes principais de uma fábrica são equipamentos, trabalhadores, lotes, etc. Podemos classificar estes componentes de acordo com o seu comportamento. Classificações gerais são locais de produção, onde os equipamentos estão situados e onde é executado trabalho, os equipamentos, que podem ser de diversos tipos consoante a função executada, e os lotes que são os clientes dos locais de produção. Estas classificações podem ser traduzidas para três classes de objectos do Smalltalk: Local, Equipamento e Lote. Cada uma destas classes é semelhante, pois descrevem objectos que têm tarefas a cumprir: um Lote pede serviço, um Equipamento proporciona serviço e um Local proporciona recursos. Em particular, um local proporciona uma zona de espera (ordenada) para quando existem mais lotes do que equipamentos para os processarem (inventário em processo de fabrico). Estas semelhanças foram agrupadas na superclasse SimulationObject, que descreve os objectos que são usados na simulação. Um SimulationObject é qualquer objecto a que pode ser atribuida uma sequência de tarefas a realizar. Cada objecto define uma sequência de actividades que são iniciadas quando o objecto entra na simulação. Por exemplo, as actividades de um lote são percorrer todas as áreas de que necessita para ser fabricado, e em cada uma delas pedir um equipamento e esperar enquanto está a ser processado. O lote entra na simulação devido a uma encomenda planeada e sai da mesma quando tiver percorrido todas as áreas necessárias.

A classe SimulationObject especifica uma sequência geral em que um objecto entra, executa as suas tarefas e sai da simulação. Esta sequência consiste em enviar ao objecto as mensagens startUp, tasks e finishUp. A inicialização das variáveis descritivas é feita como resposta à mensagem initialize. As respostas às mensagens tasks e initialize são implementadas nas subclasses de SimulationObject.

A implementação desta e doutras classes pode ser consultada no anexo D.

Existem várias mensagens que um objecto pode usar de forma a descrever as suas tarefas. Um exemplo é holdFor: periodoTempo, em que o argumento peridoTempo corresponde a uma dada quantidade de tempo em que o objecto deve

atrasar qualquer outra acção. A ideia deste atraso é a criação de um período de tempo que corresponde à execução duma determinada tarefa pelo objecto, como por exemplo o tempo que um lote demora a ser processado numa máquina.

Uma simulação pode conter recursos simples ou 'estáticos' que podem ser adquiridos pelo objecto da simulação, ou pode conter recursos coordenados (como é essencialmente o caso presente), ou seja objectos cujas tarefas necessitam de ser sincronizadas com as tarefas de outros objectos (lotes e equipamentos, por exemplo). Foram criadas mensagens que permitem ter acesso a ambos os tipos de recurso, tanto para fornecer como para adquirir os recursos.

Mais à frente referirei algumas das mensagens usadas para aceder aos recursos. Foram criadas quatro mensagens para os recursos estáticos (duas para obter o recurso, uma para produzir o recurso e uma para libertar o recurso). Foram também criadas três mensagens para os recursos coordenados (para obter, produzir e libertar o recurso).

Deve-se ter em conta, quando mais à frente se analisar os recursos, que os mesmos modelizam tipos de cliente (como lotes) e que quem pede para fornecer serviço são os 'servidores' (como equipamentos).

A obtenção de um recurso estático pode ser feita com níveis de prioridade diferentes (zero é a prioridade mínima).

Para se saber se um dado recurso existe na simulação envia-se à simulação a mensagem resourceAvailable: nomeRecurso em que nomeRecurso é o nome do recurso em que estamos interessados. Para se saber qual a quantidade de recurso existente na simulação num dado momento envia-se-lhe a mensagem inquireFor: quantidade ofResource: nomeRecurso em que verifica se existe aquela quantidade do recurso na simulação.

No caso de recursos coordenados existem mensagens para se conhecer a quantidade de servidores e clientes disponíveis num dado momento (numberOfProvidersOfResource: nomeRecurso e numberOfRequestersOf-Resource: nomeRecurso).

Também existe uma mensagem para parar a simulação que é stopSimulation.

III.3.2 Simulações

A finalidade da classe Simulation é fazer a gestão da organização dos objectos da simulação e da sequência de acções a ocorrer de acordo com o tempo simulado. As instâncias da classe Simulation mantêm uma referência a um conjunto de SimulationObjects, ao relógio da simulação e a uma fila de espera de acontecimentos à espera de serem invocados.

A unidade de tempo apropriado à simulação é guardada numa variável da instância (currentTime) que é um número em vírgula flutuante. Esta unidade pode ser segundos, minutos, horas, dias, etc. A simulação avança o relógio, pela inspeção da fila de espera, sabendo assim qual o momento em que o próximo acontecimento deve ter lugar. Se a fila de espera de acontecimentos estiver vazia, então a simulação termina.

Os objectos entram na simulação como resposta a um variado conjunto de mensagens. Dois exemplos são: scheduleArrivalOf: umaClasseObjectosSimulação accordingTo: umaDistribuiçãoProbabilidade ou scheduleArrivalOf: umObjecto-Simulação at: momento. O segundo argumento da primeira mensagem é uma instância de uma classe de distribuição de probabilidade adequada como as já mencionadas. Essa distribuição define o intervalo em que uma instância da classe de objectos que é definida como primeiro argumento é criada e enviada a mensagem startUp.

Estas mensagens são enviadas à simulação quer quando a simulação é inicializada, em resposta à mensagem defineArrivalSchedule, ou como parte da sequência de tarefas de um objecto na simulação.

A simulação também contém mensagens que fazem o planeamento de um dado conjunto de acções, como schedule: blocoAcção at: momento ou schedule: blocoAcção after: período.

Para definir os recursos da simulação pode ser enviado self produce: quantidade of: nomeRecurso, em que é criado o recurso com a quantidade respectiva (ou acrescentada caso já exista o recurso). Outra mensagem possível é self coordinate: nomeRecurso, em que nomeRecurso se refere a um recurso que será fornecido por algum (ou alguns) objecto(s) da simulação e requisitado por outros. Por exemplo, o recurso é a operação de teste, o servidor é um equipamento de teste e o cliente é um lote.

A estrutura de controlo da classe Simulation é semelhante à da classe SimulationObject. A simulação tem origem inicializando-se uma instância de Simulation e enviando-lhe a mensagem startUp. O planeamento de chegadas de objectos assim como os próprios objectos da simulação criam acontecimentos que são colocados na fila de espera. Depois de inicializada é repetidamente enviada a mensagem proceed à simulação até não existirem mais acontecimentos na fila ou não estarmos mais interessados no prolongamento da simulação.

No decurso da simulação os objectos avisam a simulação quando entram e saiem da mesma. Isto vai permitir, como veremos, a recolha de informações acerca da simulação, ou a criação de subclasses que não permitam a um objecto entrar ou sair de uma simulação sem se verificarem determinadas condições.

Várias das mensagens existentes na classe Simulation não executam nenhuma função devendo ser implementadas em subclasses que especificam a simulação respectiva. É assim possível termos vários tipos de simulações sem necessidade de alterar as classes base.

III.3.3 Implementação das classes do simulador

Para a implementação base do simulador foi necessária a criação das classes Simulation, SimulationObject e DelayedEvent.

III.3.3.1 Classe SimulationObject

Todos os SimulationObject criados no sistema necessitam de acesso à instância de Simulation a que pertencem. Esse acesso é necessário para por exemplo informar de que um objecto está a entrar ou a sair da simulação. De forma a suportar tal acesso, a classe Simulation responde à mensagem active, retornando a simulação respectiva. As tarefas de qualquer subclasse de SimulationObject enviam sempre as suas mensagens para Simulation active para por exemplo pedirem à simulação informações e obter ou produzir recursos.

O ciclo de vida de um objecto envolve a sequência início-tarefas-fim (startUp - tasks - finishUp). Logo que um SimulationObject entra na simulação é-lhe

enviada a mensagem startUp. Esta mensagem encarrega-se de após entrar na simulação de enviar a si próprio (self) as mensagens tasks e finalmente finishUp. A mensagem tasks é implementada nas subclasses de SimulationObject.

A categoria task language da classe SimulationObject contém os métodos a serem usados para especificar a sequência de acções do objecto na simulação. O objecto pode por exemplo esperar um determinado período de tempo (holdFor:), tentar obter um dado recurso (acquire:ofResource:) ou determinar quando um dado recurso deverá estar disponível (resourceAvailable:).

A classe Simulation guarda um conjunto (do tipo Set) dos recursos existentes. No caso de recursos estáticos, são guardadas instâncias da classe ResourceProvider, no caso de recursos com actividades coordenadas com outros objectos são guardadas instâncias da classe ResourceCoordinator.

Quando um SimulationObject tenta adquirir um dado recurso estático (acquire:ofResource:) e tal pedido tem sucesso, é fornecida uma instância da classe Resource ao SimulationObject. A instância de Resource está relacionada com o recurso que a criou e com a quantidade de recurso que representa. Um dado recurso responde à mensagem amountAvailable com a quantidade de recurso que o objecto pode receber. Esta mensagem é enviada no método de inquireFor:ofResource:.

Nos métodos associados às mensagens acquire:ofResource: e acquire:ofResource:withPriority: do SimulationObject, é obtido um ResourceProvider a quem é enviada a mensagem acquire: quantidade withPriority: prioridade. O resultado desta mensagem é a obtenção de uma instância da classe StaticResource. No entanto, se a quantidade de recurso não estiver disponível, o processo que fez o pedido será suspendido até que os recursos necessários fiquem disponíveis. Envia-se a mensagem release a uma StaticResource para se fazer a reciclagem de um recurso quando deixa de ser necessário a um dado objecto.

Quando um SimulationObject tenta obter um recurso coordenado (acquireResource:) e essa tentativa tem sucesso, esse objecto coopera com o objecto da simulação que age como o recurso (o objecto que necessita serviço) até as tarefas estarem prontas. Se o recurso não está disponível, o processo em que foi executado o pedido será suspendido até os recursos necessários estarem disponíveis.

As instâncias da classe ResourceCoordinator contêm as mensagens acquire (para fazerem o pedido de coordenação de tarefas) e produceBy: umObjecto (para

especificar que o objecto argumento deve ter as suas tarefas sincronizadas com outro objecto). Um ResourceCoordinator responde também a mensagens do tipo customersWaiting, ServersWaiting e queueLenght, para determinar se recursos (clientes) ou servidores estão à espera para coordenar as suas actividades e quantos estão à espera.

III.3.3.2 Classe DelayedEvent

A implementação do mecanismo de gestão (*scheduling*) da classe Simulation faz utilização intensiva das classes existentes no Smalltalk-80 que têm a ver com a gestão de processos concorrentes. Entre essas classes podem-se realçar Process, Semaphore, ProcessorScheduler, Delay e SharedQueue. Para uma melhor compreenção destas classes pode-se consultar [B13] ou [B12].

Existem três questões essenciais na implementação da classe Simulation:

- Como se guarda um acontecimento que deve ser atrasado de uma certa quantidade de tempo?
- Como nos certificamos que todos os processos que se iniciam num determinado momento se completam antes do relógio da simulação avançar?
- Como implementar a gestão permanente da sequência de acções, em particular a instanciação e inicialização dos objectos da simulação?

De forma a resolver a primeira questão, a simulação (classe Simulation) mantém uma fila de espera com os acontecimentos programados. Esta fila de espera é do tipo SortedCollection em que os elementos são os acontecimentos, ordenados pelo momento em que devem ser invocados. Cada acontecimento existe na fila como uma instância da classe DelayedEvent. No momento em que tal instância é criada o acontecimento é o que está activo na simulação, pelo que pode ser guardado com o contexto necessário através da criação de um semáforo (Semaphore). Quando um acontecimento é colocado na fila de espera, é enviada a mensagem pause ao DelayedEvent o que faz por sua vez enviar a mensagem wait ao Semaphore respectivo. Quando o acontecimento é retirado da fila de espera é--lhe enviada a mensagem resume o que por sua vez faz enviar a mensagem signal ao Semaphore do DelayedEvent.

III.3.3.3 Classe Simulation

As instâncias da classe Simulation têm quatro variáveis próprias (a classe e o nome da variável são apresentados entre parêntesis): um conjunto (Set) de objectos que actuam como os recursos da simulação (resources), um valor (Number) contendo o momento actual (currentTime), um conjunto ordenado (SortedCollection) com a fila de acontecimentos em espera (eventQueue) e um número (Integer) representando o número de processos activos no momento corrente (processCount). Uma variável de classe (RunningSimulation) mantém referência à simulação que está a correr, permitindo a existência de múltiplas instâncias de Simulation, mas apenas uma a controlar o sistema de cada vez.

Na classe Simulation existem vários métodos de forma a permitir o planeamento das várias actividades. Esses métodos são de tipo muito variado como por exemplo executar uma tarefa num dado momento (schedule: umBloco at: momento), produzir uma dada quantidade de um recurso (produce: quantidade of: nomeRecurso), planear a chegada de determinados objectos (scheduleArrivalOf: objecto at: momento), etc.

As mensagens relacionadas com a gestão de tempo (*scheduling*) acedem a um contador (processCount) que contém a quantidade de processos que estão em processo. Este é o modo de resolver a segunda questão apresentada anteriormente, ou seja garantir que o relógio da simulação só é avançado após terem terminado todas as tarefas a executar num dado momento. Desta forma garante-se que o relógio só é actualizado quando o contador for zero, o que significa já terem terminado todas as tarefas respectivas.

O método schedule: umBloco startingAt: momento andThenEvery: distribuiçãoProbabilidade, proporciona o mecanismo necessário à resolução da terceira questão posta, em que se faz o planeamento de acções repetitivas.

A ideia básica de schedule: umBloco at: momento é a criação de um processo em que se atrasa a execução de uma dada sequência de acções (umBloco) até que a simulação chegue a um determinado momento (momento). O atraso é obtido através da mensagem delayUntil: tempoAtraso. O método associado cria uma instância de DelayedEvent que é acrescentado á fila de espera da simulação. O processo associado a este acontecimento é então suspenso (através da mensagem pause). Quando esta instância de DelayedEvent for a primeira da fila de espera, será removida e o relógio da simulação será avançado para o momento respectivo.

Será então enviada a esta instância a mensagem resume o que causará a execução das tarefas do bloco, que correspondem a uma dada actividade da simulação.

A classe Simulation tem uma estrutura semelhante à classe SimulationObject já anteriormente referida. A resposta à mensagem startUp faz com que a simulação passe a estar activa, fazendo a definição dos objectos da simulação e o planeamento das suas chegadas. O ciclo interno de gestão de actividades (relembrando: startUp, tasks, finishUp) é realizado como resposta à mensagem proceed. Sempre que a simulação recebe esta mensagem, verifica o contador de processos (enviando a mensagem readyToContinue), e caso o contador não se encontre a zero, o gestor geral de processos (Processor) é solicitado a deixar evoluir os processos. No caso do contador se encontrar a zero então é verificada a fila de espera. Caso não esteja vazia, o próximo acontecimento é retirado da fila, o relógio é actualizado, e o processo reassumido.

As mensagens fundamentais em termos de gestão de processos das classes Simulation e SimulationObject, são o envio de fork a BlockContext, yield a ProcessorScheduler, e signal e wait a Semaphore. Devido a serem mensagens bem conhecidas do domínio da gestão de processos não entrarei em pormenores acerca das mesmas.

```
Object ()
        Simulation ('resources' 'currentTime' 'eventQueue' 'processCount')
        SimulationObjéct ()
        DelayedEvent ('resumptionSemaphore' 'resumptionCondition')
```

Estrutura base das classes de simulação

III.4 Recolha de estatísticas

Vou aqui apresentar a estrutura de classes responsável pela recolha de estatísticas.

Estas estatísticas podem ser de tipo muito variado, como por exemplo, o tempo que os vários objectos permanecem na simulação (e logo uma medida da eficiência do modelo relativamente às tarefas executadas pelo objecto), informação sobre as filas de espera (como máximo, mínimo, média e tempo de permanência), informação sobre a utilização dos recursos da simulação, etc.

Foram criados quatro processos de recolhas de estatísticas: estatísticas de duração, histogramas de frequência, contabilização de acontecimentos e monitorização de acontecimentos.

Cada um destes processos envolve a criação de uma subclasse de Simulation ou SimulationObject de forma a criar um dicionário e/ou um ficheiro em que se possa guardar informação, e nessa subclasse são feitas as modificações necessárias aos métodos da superclasse de forma a se guardar a informação necessária.

```
Object ()
        Simulation ('resources' 'currentTime' 'eventQueue' 'processCount')
                StatisticsWithSimulation ('statistics')
        SimulationObject ()
                EventMonitor ('label')
        Histogram ('tallyArray' 'lowerBound' 'upperBound' 'step' 'minValue'
                        'totalValues' 'extraEntries')
        SimulationObjectRecord ('entranceTime' 'duration')
```

Estrutura de classes responsável pela recolha de estatísticas

III.4.1 Estatísticas de duração

Para a recolha deste tipo de estatística foram criadas subclasses de Simulation que só necessitam de guardar o momento em que cada objecto entra e sai da simulação. Esses momentos são guardados em registos especiais que contêm o momento de entrada na simulação (entranceTime) e o tempo de permanência na mesma (duration). Esses registos são instâncias da classe SimulationObjectRecord. Um exemplo de uma subclasse criada para usar estes registos é a classe StatisticsWithSimulation.

III.4.2 Histogramas de frequência

Um tipo de estatística que é comum recolher em simulações é o histograma de frequência, ou seja, quantos objectos passam na simulação em um determinado período de tempo, o que é proporcional ao tempo que cada objecto permanece na simulação. A recolha deste tipo de estatística implica manter um registo do número de objectos que existem na simulação em períodos de tempo predeterminados.

O resultado mostra o número total de objectos, o tempo mínimo e máximo de permanência, e o número de objectos cujo tempo de permanência pertence a cada período predeterminado.

Este tipo de estatística é muito útil pois permite analisar se existem ou não recursos suficientes para responder aos pedidos de serviço existentes. Se os objectos necessitam de esperar muito tempo para serem atendidos têm de passar mais tempo na simulação. É da análise destas estatísticas que o SimAI chama a atenção para a insuficiência de equipamentos.

A classe Histogram foi criada para suportar a recolha dos histogramas de frequência. Esta classe é criada por uma mensagem onde é especificado o valor mínimo, o valor máximo e a largura de cada intervalo pretendido para o histograma. Se por exemplo pretendermos um histograma que registo o número de vezes que um dado valor se situe entre 5 e 45 em grupos de 5, a mensagem a enviar seria:

```
Histogram from: 5 to: 45 by: 5
```

Como podem existir valores que não se encaixem nos intervalos predefinido, existe uma variável (extraEntries) que contabiliza os valores que estão fora do intervalo do histograma.

Os valores nos intervalos são guardados como membros de uma matriz (*array*) cujo tamanho é igual ao número de intervalos (i.e., limite máximo - limite mínimo // largura do intervalo). A mensagem store: aValue é usada para colocar valores no histograma.

O resultado do histograma pode ser enviado para um ficheiro através da mensagem:

```
simulação printStatisticsOn: ('ficheiro.rpt' asFilename writeStream)
```

III.4.3 Contabilização de acontecimentos

Este tipo de estatística não recorre à utilização de nenhuma classe especial sendo implementada pela classe que as pretende. Consiste apenas na contabilização das variáveis pretendidas com posterior envio para um ficheiro de registo.

III.4.4 Monitorização de acontecimentos

Uma outra técnica para a recolha de dados de uma simulação é o registo da ocorrência de acontecimentos importantes, incluindo por exemplo o momento de entrada e saída de certos objectos da simulação.

Isto foi implementado pela criação de uma subclasse de SimulationObject denominada EventMonitor. Esta classe proporciona a ligação a um ficheiro em que os registos são anotados.

Cada mensagem que representa um tipo de acontecimento que desejamos registar deverá ser reescrita na subclasse respectiva de forma a incluir as instruções apropriadas ao envio de informações para o ficheiro. O método na superclasse continua disponível para execução, através do envio da mensagem para a pseudo variável super.

O registo mais usado, em termos de notações, consiste no momento e identificação do receptor (uma instância de SimulationObject), e num comentário como por exemplo "entra", "sai" ou "liberta".

Este tipo de monitorização além de ser útil para a recolha de determinadas informações, é especialmente útil na fase de desenvolvimento e depuração de erros de um programa de simulação como foi por exemplo o caso do sistema SimAI, pois permite manter um registo completo da evolução dos objectos na simulação.

III.5 Recursos

A utilização de recursos é um factor essencial em qualquer simulação. Sem a existência de recursos tudo o que um objecto pode realizar numa simulação é apenas aguardar um determinado período de tempo. Vou aqui descrever dois tipos de recursos que já foram previamente referidos: recursos fixos e recursos flutuantes.

Um recurso fixo pode ser consumível. Este tipo de recurso começa com uma determinada quantidade inicial. No decorrer da simulação, os objectos vão adquirindo parte do recurso, chegando eventualmente à sua exaustão. Novos objectos que necessitem desse recurso ou terminam sem terem completado com sucesso as suas tarefas, ou são atrasados indefinidamente esperando por um recurso que nunca lhes será proporcionado.

Um recurso fixo também pode ser não consumível. Neste caso, o recurso também começa com uma determinada quantidade inicial. No decorrer da simulação, os objectos vão adquirindo parte do recurso. Quando um objecto deixa de necessitar da quantidade de recurso que retirou é feita a reciclagem dessa parte, voltando a estar disponível. Novos objectos que necessitem desse recurso, ou o obtêm imediatamente ou são atrasados até um ser reciclado e voltar a estar disponível.

Um recurso flutuante modeliza uma relação do tipo produtor-consumidor. Uma simulação começa com uma determinada quantidade de um dado recurso. Com o decorrer da simulação os objectos vão adquirindo esse recurso, fazendo a sua reciclagem quando já não necessitarem dele. Novos recursos podem ser acrescentados em determinadas alturas, aumentando a disponibilidade desse recurso. Novos objectos que necessitem desse recurso, ou o obtêm imediatamente ou são atrasados até um ser reciclado ou um novo ser produzido. Um recurso flutuante em que quantidades adicionais podem ser produzidas é denominado recurso renovável.

Os três tipos de objectos que podemos ter na simulação (consumíveis, não consumíveis e renováveis) coordenam o acesso a recursos quantificáveis. Também é por vezes necessário coordenar as tarefas de dois ou mais objectos, por exemplo um dado equipamento proporciona serviço a um lote quando esse lote lhe é destinado.

III.5.1 Implementação das classes necessárias

Para a manipulação de recursos foram definidas as classes Resource, ResourceProvider, ResourceCoordinator, WaitingSimulationObject e Resource-Coordinator.

```
Object ()
        DelayedEvent ('resumptionSemaphore' 'resumptionCondition')
                WaitingSimulationObject ('amount' 'resource')
        Resource ('pending' 'resourceName')
                ResourceCoordinator ('whoIsWaiting')
                ResourceProvider ('amountAvailable')
        SimulationObject ()
```

Estrutura de classes responsável pelos recursos

III.5.1.1 A utilização de recursos

A classe Resource representa o recurso respectivo em termos do seu nome e da fila de espera de pedidos de serviço a serem satisfeitos.

Uma das mensagens possíveis para a criação de um recurso é:

```
self produce: quantidade of: nomeRecurso
```

A resposta a esta mensagem é a criação de uma instância da classe Resource com quantidade como a quantidade disponível do recurso, ou aceder a uma instância já existente desse recurso e aumentar a seu valor de quantidade. A classe responsável pela gestão dos recursos é ResourceProvider (subclasse de Resource) cujas instâncias são criadas enviando à classe a mensagem named: nome ou named: nome with: quantidade.

Quando um SimulationObject solicita um dado recurso (acquire: quantidade ofResource: nomeRecurso), a simulação que está no momento activa (Simulation active) é solicitada a proporcionar o referido recurso (provideResourceFor:). A simulação guarda os nomes dos recursos disponíveis numa instância de Set.

Assim que um SimulationObject tem acesso a um ResourceProvider é possível:

- Perguntar a quantidade de recurso disponível;
- Perguntar o nome do recurso;
- Pedir parte do recurso (e com uma dada prioridade);
- Pedir para produzir uma dada quantidade do recurso.

Quando um SimulationObject solicita um determinado recurso, este pedido é adicionado a uma lista de pedidos semelhantes, ordenados relativamente a prioridade, e dentro da mesma prioridade, numa ordem de chegada. Sempre que um pedido é feito ou um recurso é produzido o ResourceProvider verifica se algum dos pedidos em espera pode ser satisfeito.

Cada pedido é guardado como uma instância de WaitingSimulationObject que é uma subclasse de DelayedEvent. Sempre que um pedido é executado, esse pedido é colocado na fila de espera, pending, e é feita uma tentativa para proporcionar o recurso.

A classe ResourceProvider representa os recursos como simples items quantificáveis, que não têm tarefas a executar, e que por isso não são criados como instâncias de SimulationObject. Um contador é mantido com o número de items. Quando um SimulationObject consegue adquirir um dado recurso, é-lhe dado acesso a uma instância de WaitingSimulationObject. A resposta do ResourceProvider à mensagem acquire:withPriority: consiste em retirar o WaitingSimulationObject da fila de pedidos pendentes, decrementar a quantidade de recurso disponível no momento e retornar o referido objecto.

Um WaitingSimulationObject representa um acontecimento em espera. Pode ser por exemplo um pedido para fornecer uma dada quantidade de recurso (para ser um servidor) ou um pedido para adquirir uma dada quantidade de recurso (para ser um cliente).

Quando um SimulationObject adquire um dado recurso, mantem-no até ele ser devolvido à simulação. A quantidade de recurso que é mantida no WaitingSimulationObject é devolvida à simulação através do envio da mensagem release. Em termos de implementação, o SimulationObject envia a si próprio a mensagem release: aWaitingSimulationObject para que se crie um estilo uniforme de envio de mensagens a si próprio (self) no método associado às tarefas do objecto.

Esta uniformidade simplifica a criação de outros métodos responsáveis pela monitorização de acontecimentos. Como todas as mensagens para execução de actividades são enviadas como mensagens a self e portanto a um SimulationObject, é possível criar subclasses de SimulationObject (a classe EventMonitor, anteriormente apresentada é um bom exemplo) em que as mensagens sejam interceptadas de forma a se criarem registos do que está a acontecer.

Ao usar as mensagens acquire:ofResource: e release: de SimulationObject, o recurso pode ser tratado como um recurso não consumível. Uma mistura é também possível. O SimulationObject pode adquirir uma dada quantidade de um recurso, por exemplo 10, consumir 5 e devolver 5. A mensagem consume: é usada para retirar recursos da simulação.

III.5.1.2 Coordenação de recursos

O mecanismo que proporciona a sincronização de tarefas entre dois ou mais SimulationObject é suportado pela classe ResourceCoordinador (que é também uma subclasse de Resource).

Um ResourceCoordinator representa um SimulationObject cujas tarefas devam ser sincronizadas com as tarefas de um outro SimulationObject. Um dos objectos é considerado como o recurso ou o "cliente", e o outro objecto adquire o primeiro de forma a fornecer um dado serviço, podendo por isso ser considerado um produtor ou "servidor".

Num dado momento podemos ter clientes à espera de um servidor, servidores à espera de clientes ou ninguém à espera. Apenas é necessária a manutenção de uma fila de espera, para cada tipo de recurso. Uma variável de ResourceCoordinator mantém o controlo da fila, sabendo se a mesma contém servidores, clientes ou se está vazia. A variável pending herdada da superclasse Resource, refere-se a esta fila, a variável whoIsWaiting refere-se ao conteúdo da mesma.

A um ResourceProvider podem colocar-se três tipos de questões :

- Existem clientes à espera? (customersWaiting);
- Existem servidores à espera? (serversWaiting);
- Quantos estão à espera? (queueLenght).

A mensagem acquire vem de um SimulationObject que age como servidor e que pretende apoderar-se de um cliente de forma a servi-lo. Se um cliente está à espera, o objecto pode proporcionar-lhe serviço (giveService), caso contrário, o objecto é adicionado à fila de espera que é classificada como possuindo servidores.

A mensagem produceBy: umCliente vem de um SimulationObject que age como cliente que deseja ser servido. Se um servidor está à espera, então o objecto pode ser servido (getServiceFor: umPedidoCliente), caso contrário o objecto é adicionado à fila de espera que é classificada como possuindo clientes.

Em qualquer dos casos, a fila de espera é constituída por instâncias de WaitingSimulationObject.

III.6 Interface com o utilizador

O sistema foi desenvolvido de forma a facilitar a interface entre o utilizador e o computador. A partir de uma janela inicial de onde é possível manter o controlo da simulação fazemos surgir outras janelas para a obtenção de funções específicas. O valores introduzidos destinam-se a preencher as variáveis das classes definidas anteriormente e que constituem o núcleo do sistema simulador.

III.6.1 Controlo do programa

O controlo principal do programa é realizado através de uma janela que contém botões, que quando acionados fazem surgir outras janelas, quer para a introdução das informações necessárias, quer para o visionamento dos resultados do programa.

Esta janela encontra-se dividida em três zonas: a primeira contém os botões que acionam os écrans de entrada, a segunda contém os botões que acionam os écrans de saída e a terceira tem os botões que controlam a simulação, como por exemplo, iniciar, parar e continuar.

III.6.2 Écrans de entrada

É através dos écrans de entrada que são fornecidas ao sistema as informações que o mesmo necessita previamente ao início da simulação.

Um dos écrans permite definir as áreas existentes na fábrica e os tipos de equipamentos que existem em cada área. Quando se seleciona uma dada área, aparece a lista de tipos de equipamento associados a essa área. Quando se seleciona um dado tipo de equipamento, surgem as características do mesmo. Este écran permite introduzir, alterar ou retirar áreas, equipamentos e suas características.

Um outro écran permite definir vários períodos temporais em que estamos interessados em usar as variáveis. Cada um destes períodos corresponderá a uma

simulação no sistema, mas tal processo será transparente para o utilizador. Para cada período atribui-se um nome que servirá de referência em todos os outros écrans. Serão também introduzidas as datas de início e fim dos vários períodos. Os períodos deverão ser consecutivos e estar ordenados. A quantidade de equipamento dentro de um período é constante.

Existe um écran para a introdução nos vários períodos, do número de horas de laboração previstas em horário normal, assim como o máximo de horas extraordinários que é possível utilizar. Este écran vai permitir o cálculo correcto da quantidade de horas disponível em cada período tendo em conta dias de paragem, feriados, etc.

Um outro écran permite a entrada da quantidade de equipamentos existentes em cada área que foi previamente definida. Para cada tipo de equipamento é introduzida a sua quantidade actual e o diferença existente em cada período que foi definido.

Também existe um écran para a introdução das encomendas previstas em cada período considerado. São definidos os produtos, sendo possível definir subprodutos a partir dos anteriores. Para cada produto são introduzidos várias informações como tamanho médio do lote, tipo de fluxo de fabrico, rendimento médio da operação de fabrico, etc. Para cada produto e em cada período são introduzidas as quantidades totais que se pretende fabricar. Os valores aqui definidos servirão para gerar as distribuições de geração de lotes no simulador. A média dessa distribuição é a quantidade a fabricar a dividir pela duração do período.

Um outro écran permite a definição dos fluxos existentes na fábrica. Cada fluxo indica uma determinada sequência de equipamentos que os lotes associados a esse percurso deverão percorrer. Este quadro permite adicionar, alterar e retirar os tipos de equipamento pertencentes a cada fluxo. Também se podem acrescentar, alterar e retirar tipos de fluxo.

III.6.3 Écrans de saída

É através dos écrans de saída que é possível visionar os resultados da simulação corrida e se pretendido enviar esses resultados para uma impressora ou ficheiro em memória permanente.

Nestes écrans definimos as áreas, equipamentos, produtos e períodos de que pretendemos resultados no relatório final. Esse relatório permite obter para cada variável os valores mínimo, máximo e média, que a mesma tomou na simulação. Variáveis possíveis de analisar são tempo de processo e de espera para os produtos e além destas a utilização do equipamento e o comportamento dos inventários para as áreas.

III.6.4 Objectos essenciais

Conforme analisado no capítulo anterior uma fábrica é constituída por um conjunto de áreas de fabrico ou produção.

Cada uma destas áreas dá uma contribuição específica para a produção dos produtos a fabricar. É o fluxo a que o lote pertence que vai determinar a sequência e as áreas que ele necessita de percorrer. Por exemplo no caso de fabrico de circuitos integrados um lote com material do tipo DIP não é processado na área de *Singulate*, sendo a mesma apenas destinada aos produtos do tipo SOIC.

Cada área é constituída por conjuntos de equipamentos semelhantes. Embora a função essencial da área seja a mesma, a especificidade de certos produtos faz com que possivelmente nem todos os equipamentos existentes sejam iguais, mas estejam sim agrupados. Por exemplo no caso de fabrico de circuitos integrados a área de teste tem vários tipos de máquinas consoante o tipo de circuito a testar.

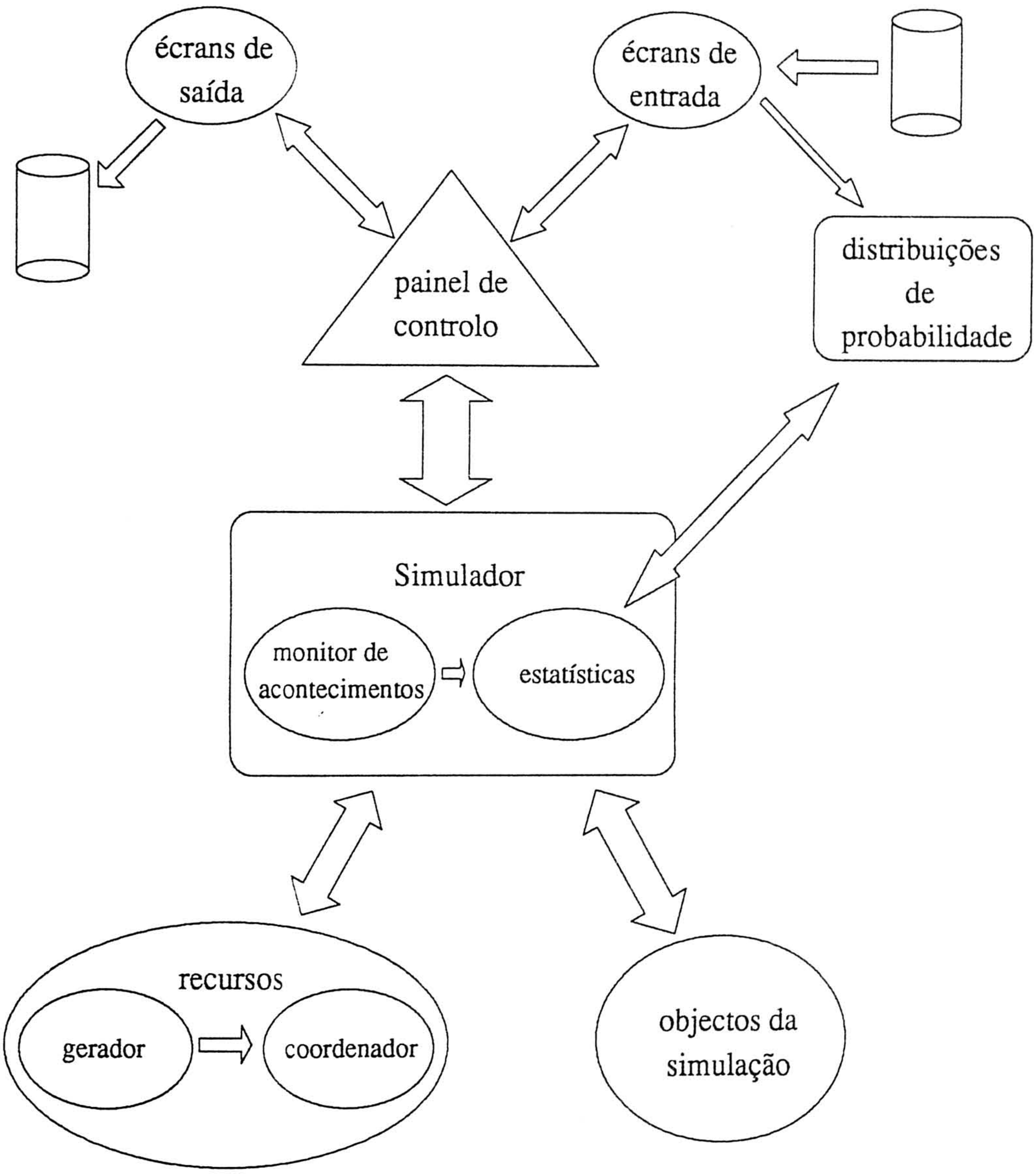
Antes de cada grupo de equipamentos existe uma fila de espera onde são colocados os lotes que pretendem obter serviço. O conjunto desses lotes constitui o inventário em processo.

Os equipamentos e os lotes são os objectos essenciais da simulação, sendo como é evidente instâncias de SimulationObject. Associados a eles existem instâncias de ResourceCoordinator para a gestão dos recursos.

Toda a simulação é efectuada da forma anteriormente descrita para o sistema base. Neste caso concreto, os vários écrans servem para a introdução dos dados que vão gerar os valores a atribuir às várias variáveis de classe e de instância. É assim possível e fácil a adaptação do sistema a qualquer ambiente deste tipo.

III.6.5 Estrutura do simulador

De acordo com o anteriormente analisado junto um esquema global do sistema SimAI.



III.7 Exemplo de aplicação

O programa desenvolvido foi aplicado ao caso concreto existente numa empresa industrial de fabrico de circuitos integrados, especificamente na Texas Instruments - Equipamento Electrónico Portugal (TIEP).

Devido a razões de segredo industrial os valores específicos aqui apresentados, como por exemplo quantidades de equipamento, velocidades de trabalho, rendimentos, volume de encomendas, etc, não correspondem à situação real.

III.7.1 Introdução

A Texas Instruments - Equipamento Electrónico Portugal (TIEP), localizada na zona da Maia, é a subsidiária em Portugal da Texas Instruments Incorporated (TI), uma companhia multinacional com mais de 50 centros de produção espalhados pela Europa, América, Ásia e Austrália.

A TIEP serve uma vasta quota do mercado europeu de semiconductores, posicionando-se no grupo das maiores empresas portuguesas sendo um dos maiores exportadores de material electrónico.

Em Portugal, desde 1973, a TI é um dos maiores empregadores na área do grande Porto, dispondo de programas de cooperação com as universidades que inclui donativos de equipamentos e atribuição de bolsas a estudantes.

As actividades da TI em Portugal são:

- Montagem e teste de circuitos integrados cobrindo uma vasta gama de circuitos digitais, lineares, controladores e memórias. Os tipos de formato (*packages*) produzidos são DIP e SOIC em 8, 14, 16, 20 e 24 pinos.
- Distribuição directa a todos os clientes europeus dos produtos produzidos em Portugal ou outros centros de produção da TI.

III.7.2 Descrição

O caso concreto adaptado da TIEP consistiu na simulação de todo o ambiente fabril, incluindo todas as áreas apresentadas no anexo C. Foram definidos os equipamentos necessários para os produtos DIP e SOIC em cada uma das áreas.

Para exemplificar uma das utilizações do sistema SimAI vamos considerar a seguinte tabela com o volume de encomendas previsto (em milhões de unidades) de 1991 a 1993 de 6 tipos de produto a fabricar.

Encomendas (MU)	1991	1992	1993
DIP 14P	60650	70260	85660
DIP 16P	90970	101410	114480
DIP 20P	50670	55930	63520
SOIC 14P	47680	56160	67520
SOIC 16P	10250	12600	15720
SOIC 20P	17380	20680	25040

Vou apresentar, na primeira tabela da página seguinte, as características dos equipamentos a usar para algumas das áreas da fábrica .

Na tabela seguinte são apresentados os fluxos correspondentes aos produtos apresentados nas áreas referidas.

Área	Equipamento	Velocidade (pph)	Rendimento (%)
DIE PREP	DSC A150	18.000	99,9
BOND	PC 3SE NB	2.100	99,8
BOND	PC 3SE WB	1.200	99,8
MOLD	SO 14P	8.400	99,9
MOLD	SO 16P	7.100	99,9
MOLD	SO 20P	6.000	99,9
MOLD	DP 14P	5.200	100
MOLD	DP 16P	5.200	100
MOLD	DP 20P	4.800	100
SINGULATE	SPC	11.000	99,8
TEST	SO NB	1.800	96,5
TEST	SO WB	1.300	96.8
TEST	DP NB	1.950	98,2
TEST	DP WB	1.760	98,1

	DIE PREP	BOND	MOLD	SINGUL.	TEST
DIP 14P	DSC A150	PC 3SE NB	DP 14P	-	DP NB
DIP 16P	DSC A150	PC 3SE NB	DP 16P	-	DP NB
DIP 20P	DSC A150	PC 3SE WB	DP 20P	-	DP WB
SOIC 14P	DSC A150	PC 3SE NB	SO 14P	SPC	SO NB
SOIC 16P	DSC A150	PC 3SE NB	SO 16P	SPC	SO NB
SOIC 20P	DSC A150	PC 3SE WB	SO 20P	SPC	SO NB

De seguida apresento uma análise da quantidade de equipamentos necessária para realizar o volume de produção anteriormente apresentado, nas referidas áreas da fábrica. É apresentada a quantidade de equipamentos actual e a necessária em cada um dos anos. As tabelas incluem os valores escolhidos com o auxílio do SimAI. Considerou-se um período anual de laboração de 221 dias de 24 horas.

Equipamentos	Área	Actual	1991	1992	1993
DSC A150	DIE PREP	2	1	1	1
PC 3SE NB	BOND	22	20	23	27
PC 3SE WB	BOND	13	12	13	15
SO 14P	MOLD	2	2	2	2
SO 16P	MOLD	1	1	1	1
SO 20P	MOLD	1	1	1	1
DP 14P	MOLD	4	3	3	4
DP 16P	MOLD	5	4	4	5
DP 20P	MOLD	3	3	3	3
SPC	SINGUL.	3	2	3	3
SO NB	TEST	7	8	9	11
SO WB	TEST	4	4	4	5
DP NB	TEST	14	15	17	20
DP WB	TEST	7	6	6	7

Nota-se quer a existência de equipamentos em excesso, como em falta (embora mais raro). Em virtude do elevado preço dos equipamentos deve-se evitar a existência de equipamentos em excesso, e em virtude dos referidos equipamentos

demorarem muito tempo a serem entregues deve-se também evitar a falta de equipamentos, para se evitar a perda de encomendas.

III.7.3 Validação dos resultados

A validação dos resultados consistiu essencialmente na comparação com a realidade existente na linha de fabrico e na utilização de alguns valores históricos existentes de forma a comprovar a validade do modelo em causa.

A situação instalada reflete os resultados apresentados pelo SimAI. Os responsáveis de produção pelas várias áreas concordaram com os resultados quer quando eles indicam excesso de equipamento, quer quando indicam a sua falta. A justificação para o excesso de equipamento foi essencialmente a falta de confiança nas ferramentas de cálculo actualmente existentes, que por vezes indicavam a quantidade de equipamentos por defeito (e por vezes por excesso) e também permitir melhor flexibilidade na gestão da produção.

A comparação com os valores históricos apenas foi possível em algumas áreas, como por exemplo a área de teste (últimos três anos) tendo os resultados sido muito próximos dos obtidos com o recurso ao sistema SimAI. A falta de alguns detalhes impediu no entanto uma verdadeira confirmação estatística.

III.7.4 Outras abordagens

III.7.4.1 Folhas de cálculo

Uma forma alternativa de efectuar este tipo de análises, sendo aliás a mais comum, consiste no uso de folhas de cálculo.

Este tipo de abordagem é menos flexível e dá resultados mais grosseiros que os obtidos por recurso a técnicas de simulação. É no entanto, habitualmente, de execução mais rápida.

No caso concreto do fabrico de circuitos integrados existem vários factores que favorecem a utilização das técnicas de simulação relativamente ao uso de folhas de cálculo:

- elevado número de variáveis (equipamentos, lotes e fluxos);
- elevado custo dos equipamentos;
- elevado tempo de entrega de equipamentos encomendados;
- necessidade de manter os custos de fabrico o mais baixo possíveis devido à forte concorrência;
- elevado custo de movimentação de equipamentos;
- utilização quase permanente dos equipamentos.

Como a utilização de técnicas de simulação permite obter resultados com um grau de confiança superior ao da utilização de folhas de cálculo, o uso de factores de confiança pode ser reduzido. Assim, especialmente nos casos limite, pode-se evitar a encomenda de um dado equipamento. Ora, como o preço dos equipamentos é elevado e a sua encomenda tem de ser efectuada com um grande período de antecedência, a compra de um dado equipamento que venha a revelar-se como não necessário terá um impacto nos custos das unidades produzidas sem o correspondente benefício.

As técnicas de simulação permitem ainda de uma forma muito mais fácil e flexível a experimentação de novas regras de produção. Devido aos elevados custos de movimentação dos equipamentos assim como à sua permanente utilização, qualquer experimentação deve ser muito bem estudada de forma teórica antes da

sua implementação para experimentação real. Este tipo de análises é mais difícil de efectuar recorrendo ao uso de folhas de cálculo.

III.7.4.2 Sistemas periciais

Recentemente têm surgido diversas tentativas de utilizar sistemas periciais como auxiliares da gestão industrial. O elevado número de variáveis envolvidas e a grande complexidade organizacional existente, especialmente em casos como aquele em que se testou este trabalho, augura para estes sistemas um bom domínio de aplicação.

Tais sistemas existem actualmente apenas associados à gestão de produção e não ainda largamente difundidos.

Em domínios como o deste trabalho, o recurso a técnicas de simulação é mais adequado, podendo ser enriquecidas, em alguns casos, pela utilização de técnicas de inteligência artificial, em especial no domínio dos sistemas periciais.

III.7.4.3 Outros sistemas simuladores

O uso de simuladores para representar ambientes de produção ainda não se encontra largamente difundido. A maioria dos sistemas existentes são pouco flexíveis pois estão fortemente orientados para um caso específico.

Existe no entanto uma companhia americana, CACI, que tem desenvolvido trabalho interessante nesta área. Eles desenvolveram uma linguagem de programação denominada SIMSCRIPT II, orientada para a modelização de situações tendo em vista a sua posterior simulação.

Existem várias aplicações que foram desenvolvidas em SIMSCRIPT II, como por exemplo NETWORK II para simulação de redes de comunicações entre computadores, COMNET II para simulação de redes de voz, SIMLAN II para simulação de redes locais, e finalmente SIMFACTORY II para simulação de fábricas. Estas aplicações não necessitam de programação. Como se vê das aplicações existentes, têm como especial desvantagem a sua pouca flexibilidade.

Só para simular redes de comunicação existem três tipos de aplicações dependendo da sua especificidade.

Muito recentemente, esta mesma companhia, rendida às grandes vantagens que as linguagens orientadas por objectos proporcionam, criou uma nova linguagem denominada MODSIM II que se destinará no futuro a substituir o SIMSCRIPT II. Ainda não existem quaisquer aplicações escritas nesta linguagem.

Uma vantagem desta linguagem é a existência de SIMGRAPHICS II, que é uma aplicação gráfica (ainda não desenvolvida para o MODSIM II) que permite facilitar a representação visual de uma simulação.

A principal desvantagem das linguagens ou aplicações referidas são o seu elevado preço que é de cerca de US$ 30.000 cada, para plataformas do tipo computador pessoal, podendo ultrapassar muito este preço para computadores de porte mais elevado.

Por uma muito breve comparação que realizei entre o Smalltalk e o SIMSCRIPT II e o MODSIM II, fiquei convencido que a maior pureza do Smalltalk em termos de orientação por objectos assim como a sua maior generalidade, lhe conferem vantagens a longo prazo, embora seja uma linguagem menos orientada para a aplicação em simulações o que obrigou a desenvolver as várias rotinas de base de uma simulação.

Por uma comparação semelhante realizada entre o sistema desenvolvido e o SIMFACTORY II, verifica-se que o sistema comercial tem mais opções para modelização de forma automática, embora seja menos flexível em termos globais. Para os casos mais frequentes é mais fácil criar uma simulação no sistema desenvolvido do que no sistema comercial. A possibilidade de acesso para alteração ao código do sistema desenvolvido é uma vantagem pois permite a sua fácil adaptação a situações invulgares. A representação gráfica é uma vantagem do SIMFACTORY II para alguns casos particulares.

O sistema foi desenvolvido com um custo muito inferior ao que seria com estes produtos, embora à custa de mais tempo.

III.7.4.4 Comparação com abordagens alternativas

Apresento de seguida a quantidade de equipamentos que são necessários para produzir um determinado volume de encomendas (exemplo já anteriormente apresentado) com a quantidade a que se chegou usando a folha de cálculo anteriormente referida e entre parêntesis os valores recomendados com o auxílio do SimAI.

Equipamentos	Área	Actual	1991	1992	1993
DSC A150	DIE PREP	2	2 (1)	2 (1)	2 (1)
PC 3SE NB	BOND	22	20 (20)	23 (23)	27 (27)
PC 3SE WB	BOND	13	12 (12)	14 (13)	16 (15)
SO 14P	MOLD	2	2 (2)	2 (2)	2 (2)
SO 16P	MOLD	1	1 (1)	1 (1)	1 (1)
SO 20P	MOLD	1	1 (1)	1 (1)	1 (1)
DP 14P	MOLD	4	3 (3)	3 (3)	4 (4)
DP 16P	MOLD	5	4 (4)	4 (4)	5 (5)
DP 20P	MOLD	3	3 (3)	3 (3)	3 (3)
SPC	SINGUL.	3	3 (2)	4 (3)	4 (3)
SO NB	TEST	7	7 (8)	9 (9)	11 (11)
SO WB	TEST	4	3 (4)	4 (4)	5 (5)
DP NB	TEST	14	15 (15)	17 (17)	20 (20)
DP WB	TEST	7	6 (6)	7 (6)	7 (7)

Nota-se a existência de algumas diferenças entre os dois resultados, o que vem confirmar o conhecimento empírico de que os valores obtidos através da folha de cálculo apresentam por vezes resultados em excesso e por vezes resultados por defeito.

Isto acontece devido à complexidade de fluxos de fabrico que são apenas contemplados parcialmente na folha de cálculo, mas que o simulador tem em conta. Nos equipamentos que processam apenas um tipo de produto, como é o caso de molde os resultados são semelhantes.

Evitar a compra (ou atrasá-la) de um equipamento de *test* ou de *die prep*, por exemplo, significa várias dezenas de milhares de contos.

Bastaria evitar a compra de um equipamento para justificar uma ferramenta como o SimAI.

O SimAI permite também analisar o comportamento dos inventários existentes em cada área, o que não é possível presentemente, permitindo experimentar soluções que levem à sua redução.

Conclusões

IV. Conclusões

Pretendi com o meu trabalho, não só implementar uma ferramenta útil para aplicação no sector industrial, como contribuir com algo de novo e em domínios da investigação recentes, como é o caso da programação orientada por objectos e da aplicação da simulação à gestão de processos.

A escolha da linguagem Smalltalk revelou-se acertada apesar das muitas dificuldades encontradas. O estilo de programação do Smalltalk é muito diferente das linguagens mais tradicionais, obrigando a um tipo de raciocínio muito diferente. O facto de existir um número muito reduzido de pessoas com prática nesta linguagem, obrigou-me à sua aprendizagem sózinho. O esforço foi no entanto recompensado, podendo eu agora começar a tentar difundir esta linguagem quer na empresa onde trabalho, quer no meio académico local.

O sistema SimAI, desenvolvido nesta tese, tem uma utilidade prática real, tendo atingido claramente os objectivos a que inicialmente me propus.

IV.1 Melhorias futuras

O sistema desenvolvido provou ser eficaz para o objectivo pretendido. Existem no entanto, sempre alguns pontos em que se pode melhorar um sistema. Os planos para a melhoria deste sistema incluem:

- Criação de ajuda automática (*help*) em cada janela de interacção com o utilizador, sensível ao contexto existente.

- Pormenorizar o modelo fabril de modo a incluir pessoas e espaço ocupado. Embora estes parâmetros sejam facilmente calculados com os resultados da aplicação desenvolvida, a introdução destes cálculos reduz trabalho posterior.

- Criar uma representação gráfica, que mostre numa janela a evolução dos lotes dentro do ambiente fabril.

- Criar uma opção no menu principal que permita alterar a linguagem a aparecer nos menus, mensagens e relatórios. As linguagens iniciais seriam o português e o inglês, mas qualquer outra linguagem seria facilmente acrescentada recorrendo a um dicionário interno.

- Tentar passar a aplicação desenvolvida para Smalltalk/V. Esta alteração permitiria correr a aplicação em microcomputadores mais acessíveis. Tenho no entanto algumas dúvidas, que tais computadores tenham suficiente poder de processamento para realizar este tipo de simulações sem se tornarem demasiado lentos.

forsan et haec olim meminisse juvabit

Bibliografia

[B1] Averill M. Law & W. David Kelton
Simulation Modeling and Analysis
McGraw Hill, 1982

[B2] Eurico Carrapatoso
Simulação - apontamentos
FEUP, 1988

[B3] Carlos Moreira da Silva
Simulação - apontamentos
FEUP

[B4] G. S. Fishman
Principles of Discrete Event Simulation
Wiley, 1978

[B5] Joseph Talavage
Simulating Manufacturing Systems
artigo no livro Computers and Manufacturing Productivity
IEEE Press, 1987

[B6] Paulo O. R. Monteiro
Simulação de um Ambiente de Manufactura
FEUP, 1989

[B7] BYTE, V/10 N/10 October 1985
10 artigos sobre Simulação

[B8] Tom Manuel
Object Oriented Programming: a star is born
Electronics, May 1989

[B9] Brad J. Cox
Object Oriented Programming - An Evolutionary Approach
Addison-Wesley, 1987

[B10] BYTE, V/15 N/10 October 1990
8 artigos sobre Programação por objectos

[B11] IEEE Expert, V/5 N/6 December 1990
Object Oriented Programming

[B12] Adele Goldberg & David Robson
Smalltalk-80, The Language
Addison Wesley, 1989

[B13] Objectworks for Smalltalk-80 V/2.5
Tutorial Guide
User's Guide
Advanced User's Guide
Reference Guide
ParcPlace Systems, 1990

[B14] Smalltalk/V286 - Tutorial and Programming Handbook
Digitalk Inc., 1988

[B15] Goodies #1, #2, #3 for Smalltalk/V
Digitalk Inc., 1987

[B16] BYTE, V/6 N/8 August 1981
13 artigos sobre Smalltalk

[B17] Carl Philips
Teach Yourself Smalltalk
PCW, Jan/Feb/Mar 1988

[B18] Jakob Nielsen & John T. Richards
The Experience of Learning and Using Smalltalk
IEEE Software, May 1989

[B19] Jean-Paul Trembley & Paul G. Sorenson
An Introduction to Data Structures with Applications
McGraw-Hill, 1986

[B20] R. Assis e V. Távora
Projectos Industriais
Lidel, 1988

Anexos

Índice

Anexo A:	Smalltalk	99
Anexo B:	Estatística	106
Anexo C:	Montagem e teste de circuitos integrados	113
Anexo D:	Classes da simulação	121

A. Smalltalk

A.1 Um pouco de história

Nos inícios dos anos setenta o Xerox Palo Alto Research Center Learning Group iniciou um trabalho tentando criar uma visão de qual a melhor forma para que diferentes pessoas pudessem usar de forma eficiente e agradável o poder dos computadores. Em 1981 o nome do grupo foi alterado para Software Concepts Group. O seu objectivo era criar um poderoso sistema de informação, em que o utilizador pudesse guardar, aceder e manipular informação, de forma a que o sistema pudesse crescer à medida que as ideias fossem crescendo. Com o conhecimento do utilizador em como usar o sistema o número e os tipos de diferentes componentes poderiam aumentar.

Esta estratégia concentrou-se em duas áreas principais de investigação. Uma linguagem descritiva (linguagem de programação) que serviria de interface entre os modelos na mente do utilizador e a sua representação em termos do hardware disponível, e uma linguagem interactiva (uma interface de utilização) que fizesse corresponder o sistema de comunicação dos seres humanos àquele do computador.

Esta investigação tem seguido um ciclo de entre dois a quatro anos. Cria-se um sistema incorporando a compreensão actual das necessidades em termos de software, geram-se aplicações que testam a capacidade do sistema em as suportar, e finalmente, como resultado dessa experiência, reformula-se a compreensão das necessidades em termos de software e volta-se a projectar a linguagem de programação e/ou a interface com o utilizador.

Esta investigação ainda continua. A primeira versão comercial foi o Smalltalk-72 lançado em 1972, sendo o Smalltalk-80 a quinta versão deste ciclo. A partir daí manteve-se o nome e neste momento o sistema está na sua versão 2.5, em que foi desenvolvido o trabalho desta tese, mas a versão 4.0 já foi anunciada para comercialização breve, passando o nome para apenas Smalltalk. O Smalltalk/V produzido pela Digitalk baseia-se no Smalltalk-80 e destina-se a computadores menos potentes.

A.2 Smalltalk e outras linguagens convencionais

Vou aqui apresentar uma visão geral de alguns exemplos de código em Smalltalk e em Pascal para ajudar a perceber mais facilmente esta linguagem.

Os exemplos seguintes constituem uma série de expressões em Pascal (na coluna da esquerda) e em Smalltalk (na coluna da direita).

A.2.1 Afectação a uma variável escalar

```
a := b + c
```

```
a := b + c    ou    a <- b + c
```

O operador de afectação é em ambos os casos o sinal := . Em Smalltalk-80 é também usual o emprego do sinal <- . Esta expressão pode ser escrita da mesma forma em Pascal e em Smalltalk. No exemplo, o conteúdo da variável b é somado ao conteúdo da variável c e o resultado colocado em a. Em Pascal o valor calculado é guardado, enquanto em Smalltalk este tipo de expressão guarda um apontador para um objecto que contém o resultado.

A.2.2 Uma sequência de expressões

```
x := 0;
y := 'nome';
z := w
```

```
x := 0.
y := 'nome'.
z := w
```

O separador de expressões é o ponto e vírgula (;) em Pascal e o ponto final (.) em Smalltalk. Note-se que em ambas as linguagens não é usado separador na última expressão da série. Em ambas as linguagens uma *string* é um *array* de caracteres.

A.2.3 Chamar uma função com um argumento

```
a := size(array)
```

```
a := array size
```

A função size é chamada com o argumento array e o valor retornado colocado na variável a. Em Smalltalk, chamar uma função é conhecido como enviar uma mensagem. Neste caso a mensagem size é enviada ao conteúdo da variável array.

A.2.4 Chamar uma função com dois argumentos

```
x := max(x1, x2);                    x := x1 max: x2.
y := sum(p,q)                        y := p + q
```

Em Pascal os argumentos da chamada à função encontram-se dentro de parêntesis. Em Smalltalk, numa mensagem com dois argumentos, os argumentos precedem e seguem o nome da mensagem. Note-se que em Smalltalk as operações aritméticas habituais são executadas através do envio de mensagens. No primeiro caso a mensagem max: é enviada ao conteúdo da variável x1 (o primeiro argumento) , com o conteúdo de x2 como o segundo argumento. O resultado é afectado à variável x. No segundo exemplo, a mensagem + é enviada ao conteúdo da variável p com o conteúdo da variável q como o segundo argumento, e o resultado devolvido é afectado à variável y.

A.2.5 Chamar uma função com três argumentos

```
b := between(x, x1, x2)              b := x between: x1 and: x2
```

Em Smalltalk quando uma mensagem tem três ou mais argumentos, o nome da mensagem é dividido em partes, e depois do primeiro argumento, cada parte do nome da mensagem precede os argumentos seguintes. Esta distribuição do nome da mensagem torna mais claro o objectivo da mesma, ajudando a descrever os argumentos. No exemplo, o nome da mensagem é between:and: e os argumentos as variáveis x, x1 e x2. Neste exemplo testa-se se a variável x se encontra entre os valores contidos nas variáveis x1 e x2, e afecta-se o resultado *booleano* (verdadeiro ou falso) à variável b.

A.2.6 Acesso a variáveis indexadas

```
x := a[i];                           x := a at: i.
a[i + 1] := y;                       a at: i + 1 put: y.
a[i + 1] := a[i]                     a at: i + 1 put: (a at: i)
```

No Pascal usam-se os parêntesis rectos para especificar a indexação, enquanto o Smalltalk usa as mensagens at: e at:put:. No primeiro exemplo o valor da

variável i é usado para indexar um *array* identificado pela variável a, e o valor obtido é guardado na variável x. No segundo exemplo substitui-se um elemento de um *array* por um novo valor. No terceiro exemplo acede-se e alteram-se elementos de um *array*. Em Smalltalk usam-se parêntesis para indicar a ordem de cálculo.

A.2.7 Expressões condicionais

```
if a < b then a := a + 1;
```

```
a < b ifTrue: [a := a + 1]
```

```
if atEnd(ficheiro)
        then reset(ficheiro)
        else c := next(ficheiro)
```

```
steam atEnd
        ifTrue: [ficheiro reset]
        ifFalse: [c := ficheiro next]
```

Ambas as linguagens proporcionam capacidades semelhantes para a execução de expressões condicionais baseadas no resultado do cálculo de uma expressão *booleana*. Em Smalltalk as expressões condicionais estão dentro de parêntesis rectos. No primeiro exemplo, a variável a será incrementada de uma unidade se o valor dessa variável for inferior ao valor da variável b. O segundo exemplo mostra uma forma de aceder a um ficheiro, identificado pelo nome ficheiro. Se o ficheiro se encontra no fim, a mensagem reset é enviada para o reposicionar no início, se não, o próximo caracter do ficheiro é atribuido à variável c.

A.2.8 Expressões iteractivas

```
while i < 10 do
    begin
        sum := sum + a[i];
        i := i + 1
    end;
```

```
[i < 10]
    whileTrue:
        [sum := sum + (a at: i).
         i := i + 1].
```

```
for i:= 1 to 10 do
        a[i] := 0
```

```
1 to: 10 do: [ :i |
        a at: i put: 0]
```

Ambas as linguagens proporcionam a repetição de um dado conjunto de instruções. No primeiro exemplo as duas expressões do ciclo serão executadas enquanto o valor da variável i for inferior a 10. No segundo exemplo a variável i toma valores de 1 a 10 sucessivamente.

A.2.9 Retorno de valores em funções

```
functionName :=                          ^resposta
        resposta;
return
```

Em Pascal o resultado de uma função é associado ao nome dessa função, que serve como variável para conter o resultado. Em Smalltalk o caracter correspondente ao acento circunflexo (Smalltalk/V) ou uma seta na vertical (Smalltalk-80) aparece antes da expressão que é o resultado da função (ou método, como é designada em Smalltalk). Este sinal causa uma paragem na execução do método, e o valor da expressão é devolvido como resultado do mesmo. No exemplo acima o valor da variável resposta é o resultado da função.

A.2.10 Afectação e recuperação de espaço de memória

```
new(p)                                   p := Array new: 5
dispose(p)
```

A criação dinâmica de variáveis (em terminologia Smalltalk, objectos) é permitida em ambas as linguagens. No exemplo acima, ambas as linguagens afectam à variável p um apontador para o novo objecto. No entanto em Pascal é necessário reclamar o espaço ocupado por apontadores que já não são necessários para libertar memória. Em Smalltalk, a libertação de espaço de memória (*garbage collection*) é automática. Isto simplifica a programação e liberta de uma possível fonte de erros.

A.2.11 Exemplo de um programa

Segue-se um pequeno programa escrito em ambas as linguagens. O programa será escrito em Smalltalk num estilo de programação do tipo Pascal. Posteriormente será reescrito, aplicando as vantagens das características de programação por objectos da linguagem, de forma a reduzir de forma significativa a quantidade de código necessário, sem no entanto tornar a sua leitura menos explícita.

```
program frequency;
size := 80;
var   s: string[size];
      i: integer;
      c: char;
      f: array[1..26] of integer;
      k: integer;
begin
writeln('escreva linha');
readln(s);
for i := 1 to 26 do
      f[i] := 0;
for i := 1 to size do
      begin
      c := asLowerCase(s[i]);
      if isLetter(c) then
            begin
            k := ord(c)
                - ord('a')
                + 1;
            f[k] := f[k] + 1
            end
      end;
for i := 1 to 26 do write(f[i], ' ')
end.
```

```
| s c f k |

f := Array new: 26.

s := Prompter prompt: 'escreva linha'
                default: ' '.

1 to: 26 do: [ :i |
      f at: i put: 0].
1 to: s size do: [ :i |

c := (s at: i) asLowerCase.
c isLetter
      ifTrue [
            k := c asciiValue
                - $a asciiValue
                + 1.
            f at: k put: (f at: k) + 1
            ]
      ].
^f
```

Este exemplo mostra as semelhanças existentes entre a sintaxe da linguagem Pascal e da linguagem Smalltalk. O algoritmo usado é o mesmo em ambos os casos. Os caracteres de entrada são examinados um a um, e caso pertençam ao alfabeto, o contador de frequência da letra respectiva é incrementado de uma unidade.

Nenhum dos blocos previamente definidos no Smalltalk foram usados. O programa será agora reescrito de forma a usar estes blocos.

```
| s f |
s := Prompter prompt: 'escreva linha' default: ' '.
f := Bag new.
s do: [ :c := c isLetter ifTrue: [f add: c asLowerCase]].
^f
```

Um Prompter é usado para se obter a frase de entrada do utilizador. Um *prompter* é um tipo especial de janela (*window*). Um Bag vazio é criado para conter as frequências dos caracteres. Os *bags* são um tipo de conjunto (collection) que conta a ocorrência dos objectos que possui. A frase de entrada é então examinada. Se um dado caracter for do tipo letra então o seu equivalente em letra minúscula é adicionado ao conjunto. O conjunto resultante é então devolvido.

O exemplo acima dá já uma ideia do sistema poderoso que o Smalltalk implementa. Apenas com o uso de um número reduzido de objectos já definidos foi possível reescrever o programa de uma forma muito mais simples. O Smalltalk já vem com algumas centenas de classes de tipos de objectos, que por sua vez contêm vários milhares de métodos que se podem usar. Cada classe ou objecto que o utilizador cria, fica ligado ao sistema como fazendo parte dele de forma a também poder ser usado no futuro em novos programas.

B. Estatística

B.1 Definições

Numa simulação uma tentativa (*trial*) corresponde ao avanço do relógio da simulação, que pode representar segundos, minutos, horas ou qualquer outro período de tempo apropriado à simulação em causa. Um acontecimento (*event*) corresponde por exemplo ao acto de um lote chegar a uma máquina para ser processado.

A probabilidade de um acontecimento ocorrer é tipicamente obtida pela observação de um número elevado de tentativas reais. Em geral, vários acontecimentos podem tomar lugar durante um certo período de tempo. O conjunto dos vários acontecimentos possíveis é denominado espaço de amostragem (*sample space*).

A função de probabilidade de um espaço de amostragem é definida como a associação de um número entre 0 e 1 a cada acontecimento do espaço de amostragem. A probabilidade de pelo menos um dos acontecimentos do espaço de amostragem ocorrer é definida como 1. Se p for a probabilidade de um dado acontecimento A ocorrer, então a probabilidade de que A não ocorra é definida por $(1-p)$.

Existem dois tipos de espaços de amostragem: discretos e contínuos. Um espaço de amostragem diz-se discreto se contém um número finito de acontecimentos possíveis, por exemplo os seis acontecimentos possíveis de obter com o lançamento de um dado constituem um espaço de amostragem discreto. Um espaço de amostragem diz-se contínuo se contém um número infinito e ordenado de elementos, por exemplo qualquer número entre 1.0 e 4.0. As funções de probabilidade que se aplicam a cada um destes tipos são denominadas respectivamente funções de probabilidade discretas e funções de probabilidade contínuas.

Uma variável aleatória (*random variable)* é uma função de valor real definida sobre os acontecimentos de um espaço de amostragem. Os adjectivos discreta ou contínua aplicam-se a uma variável aleatória de acordo com as características do seu intervalo (*range*). A função de probabilidade de uma variável aleatória

é denominada distribuição de probabilidade. Os valores no intervalo da função correspondem à probabilidade de ocorrência dos valores possíveis de tomar pela variável aleatória. A densidade é uma função que afecta probabilidades ao intervalo permitido da variável aleatória. Qualquer função pode ser uma função de densidade (discreta ou contínua) se apenas possuir valores positivos e cujo integral seja 1.

Outra função importante é denominada função de distribuição cumulativa. Ela dá-nos a probabilidade de que o valor de uma variável aleatória se encontre dentro de um dado intervalo.

A média é definida como o valor esperado da variável. A variância é uma medida do espalhamento da função em torno da média e é definida como o valor esperado do quadrado dos desvios em torno da média.

B.2 Distribuições de probabilidade

B.2.1 Distribuições discretas

Vou definir quatro tipos de distribuições: Bernoulli, Binomial, Geométrica e Poisson.

A distribuição de Bernoulli responde a questões como "Teremos um sucesso na próxima tentativa?".

A distribuição binomial representa a distribuição de Bernoulli repetida N vezes de forma independente, em que N é maior or igual a um. Responde a questões como "Quantos sucessos teremos nas próximas N tentativas?".

A distribuição geométrica faz uma abordagem ligeiramente diferente, e responde a questões como "Quantas tentativas independentes do tipo Bernoulli serão necessárias até o primeiro sucesso ser obtido?".

A distribuição de Poisson responde a questões do tipo "Quantos acontecimentos ocorrerão num determinado intervalo de tempo?". Em particular a distribuição de Poisson determina a probabilidade de K acontecimentos ocorrerem num dado intervalo de tempo, em que K é maior ou igual a zero.

B.2.1.1 Distribuição de Bernoulli

A distribuição de Bernoulli é usada no caso de um espaço de amostragem de apenas duas possibilidades, cada uma com uma dada probabilidade de ocorrência. Exemplos deste tipo de espaços de amostragem são:

- Atirar um dado, em que a questão é "Saiu a face 4?". A probabilidade de sucesso no caso do dado não estar viciado é 1/6 e a probabilidade de não sucesso é 5/6.

- Atirar uma moeda, em que a questão é "Saiu cara?" A probabilidade de sucesso no caso da moeda não estar viciada é 1/2 e a probabilidade de não sucesso também é 1/2.

- Retirar uma carta de um baralho em que a questão é "Saiu a rainha de espadas?". A probabilidade de sucesso num baralho normal é de 1/52 e a probabilidade de não sucesso é 51/52.

Na distribuição de Bernoulli a média é igual à probabilidade de sucesso, pelo que os dois termos são por vezes usados indistintamente.

B.2.1.2 Distribuição Binomial

A distribuição de Bernoulli é usada em simulações para responder a questões do tipo "Será que um lote vai chegar a este equipamento na próxima unidade de tempo?" ou "Será que teremos uma avaria nesta máquina hoje?". A distribuição binomial permite calcular quantos sucessos poderemos esperar em N tentativas. A função densidade de probabilidade de uma distribuição de Bernoulli dá-nos a probabilidade de ocorrência de um de dois possíveis acontecimentos. A função densidade de probabilidade da distribuição binomial dá-nos a probabilidade de termos X sucessos nas próximas N tentativas.

A distribuição binomial representa a repetição de N tentativas de Bernoulli independentes. O seu resultado é o mesmo da distribuição de Bernoulli para N = 1.

A função de probabilidade para a distribuição binomial é:

$$\frac{N!}{x!\,(N - x)!}\,p^x\,(1 - p)^{N - x}$$

em que x é o número de sucessos e p é a probabilidade de sucesso em cada tentativa. A notação "!" representa, como é habitual, a função matemática factorial.

B.2.1.3 Distribuição Geométrica

A distribuição geométrica dá-nos o número de tentativas do tipo Bernoulli que são necessárias até o primeiro sucesso ser obtido. A média da distribuição geométrica é igual ao inverso da probabilidade de sucesso.

A distribuição geométrica é mais usada em simulações orientadas por acontecimentos do que as distribuições de Bernoulli e binomial. Em vez de calcular quantos lotes chegarão nos próximos 5 minutos (uma pergunta do tipo binomial), a distribuição geométrica calcula quantos minutos decorrerão até ao próximo lote chegar. Nas simulações orientadas por acontecimentos o relógio da simulação avança para o momento do próximo acontecimento. Usando uma distribuição geométrica, pode-se determinar quando é que o próximo acontecimento terá lugar, ajustar o relógio de acordo, e então fazer ocorrer o acontecimento, poupando potencialmente muito tempo.

A função de distribuição de probabilidade é:

$$p\,(1 - p)^{x - 1}$$

em que x é o número de tentativas necessário e p é a probabilidade de sucesso de uma tentativa.

B.2.1.4 Distribuição de Poisson

Em simulações, a distribuição de Poisson é útil para realizar amostragens de potenciais pedidos de serviço por parte de clientes, como por exemplo de lotes a equipamentos numa fábrica. A prática demonstra que a taxa a que o serviço é prestado se aproxima de uma distribuição de Poisson.

A distribuição de Poisson descreve a probabilidade de que ocorram exactamente x acontecimentos na próxima unidade de tempo, em que a média de ocorrência por unidade de tempo é a variável μ. Para um dado intervalo de tempo dt, a probabilidade é $\mu * dt$; μ deve ser maior que 0.

A função de distribuição de probabilidade é:

$$\frac{a^x e^{-\mu}}{x!}$$

em que μ é a média de ocorrência, *e* é a base dos logarítmos naturais, *x* é o número de ocorrências e *!* a notação de factorial.

A função densidade de probabilidade de x determina a probabilidade de x acontecimentos ocorrerem na próxima unidade de tempo . A função de distribuição cumulativa de x determina a probabilidade de numa unidade de tempo acontecerem x ou menos acontecimentos.

B.2.2 Distribuições contínuas

Uma variável aleatória contínua pode assumir qualquer valor num intervalo ou num conjunto de intervalos. Existe uma forte correspondência entre o tipo de questões postas no caso contínuo com a caso discreto. Um exemplo de um tipo de resultado que se pode obter de uma distribuição de probabilidade contínua é qual é a probabilidade de se ter uma dada temperatura num dado momento. A temperatura é uma propriedade física que é medida numa escala contínua.

Vou definir quatro tipos de distribuições: Uniforme, Exponencial, Gamma e Normal.

A distribuição uniforme responde a questões do tipo "Tendo um grupo de acontecimentos igualmente prováveis, qual é que ocorrerá?".

A distribuição exponencial parte do princípio que os acontecimentos estão distribuídos segundo uma distribuição de Poisson e responde a questões do tipo "Quanto tempo decorrerá até que o próximo acontecimento se dê?".

A distribuição gamma está relacionada com a exponencial e responde a questões do tipo "Quanto tempo decorrerá até que tomem lugar os próximos N acontecimentos?".

A distribuição normal ou Gaussiana é muito útil para aproximar as formas limites das outras distribuições. Tem um papel importante na estatística devido a ser muito simples de usar, é simétrica relativamente à média, completamente determinada por apenas dois parâmetros (a média e a variância) e reflecte a distribuição da maioria dos acontecimentos usuais.

B.2.2.1 Distribuição Uniforme

A distribuição unifome já foi analisada na perspectiva de selecionar elementos discretos de um espaço de amostragem finito. A questão colocada era a de selecionar um acontecimento de entre vários com igual probabilidade. No caso contínuo o espaço de amostragem é um *continuum*, como por exemplo tempo ou o intervalo entre 0 e 1.

B.2.2.2 Distribuição Exponencial

Sabendo que os acontecimentos estão distribuídos de acordo com a lei de Poisson, a distribuição exponencial determina quanto tempo decorrerá até o próximo acontecimento ocorrer. Esta aproximação é mais útil no caso de simulações, da mesma forma que a distribuição geométrica é mais útil que a distribuição binomial, pois permite fazer saltar o relógio da simulação para o momento da ocorrência do próximo acontecimento, em vez de se ter de inspecionar cada unidade de tempo.

Um exemplo da utilização da distribuição exponencial é a questão "Quando é que chegará o próximo lote a este equipamento?". A função densidade dá a probabilidade de o próximo acontecimento ocorrer no intervalo de tempo x, como por exemplo "Qual é a probabilidade de o próximo lote chegar dentro de 10 minutos?".

A distribuição exponencial é usada habitualmente em situações em que a amostra é deteriorável com o tempo. Por exemplo no cálculo da probabilidade de que uma lâmpada ou um dado equipamento electrónico falhe antes de um dado momento x. A distribuição exponencial é útil neste tipo de casos, em que quanto maior for o tempo em que o equipamento se encontra a funcionar, maior é a probabilidade de ele avariar.

Tal como no caso da distribuição de Poisson, o parâmetro da distribuição exponencial, μ, é medido em termos de acontecimentos por unidade de tempo, embora o domínio da distribuição seja tempo e não acontecimentos.

A função de probabilidade da distribuição exponencial é:

$$\frac{e^{-\frac{x}{a}}}{a}$$

em que *a* é a taxa média ($\mu = 1/a$) entre ocorrências.

B.2.2.3 Distribuição Gamma

A distribuição gamma está relacionada com a exponencial. Usa-se a distribuição gamma para saber por exemplo quanto tempo decorrerá até o N-ésimo lote chegar a uma dada máquina.

A função de probabilidade é: $$\frac{x^{k-1} e^{-\frac{x}{a}}}{a^k (k-1)!}$$

em que k é maior que zero e o parâmetro $\mu=1/a$. O segundo termo do denominador, (k-1)!, é a função gamma.

B.2.2.4 Distribuição Normal ou de Gauss

A distribuição normal, também conhecida como gaussiana, é útil para sumariar ou aproximar outras distribuições. Usando a distribuição normal podem-se colocar questões como quanto tempo decorrerá antes que um sucesso ocorra (tal como na distribuição binomial), ou quantos acontecimentos ocorrem num dado intervalo de tempo (tal como na distribuição de Poisson), etc. Na realidade, a distribuição normal pode ser usada para aproximar a distribuição binomial quando o número de acontecimentos é muito grande ou a distribuição de Poisson quando a média é grande. No entanto esta aproximação só é correcta nas regiões próximas da média, aumentando o erro da aproximação quando nos distanciamos desse valor.

A distribuição normal é usada quando existe um valor central dominante (a média), e a probabilidade de obter um dado valor diminui quando nos desviamos da média. O desenho da função de probabilidade no plano tem a forma de um sino, centrado no valor médio. Esta forma tem a ver com a necessidade de simetria em torno da média, a infinidade de valores possíveis no espaço de amostragem, e de no entanto a soma da probabilidade de todos esses valores ter de ser 1.

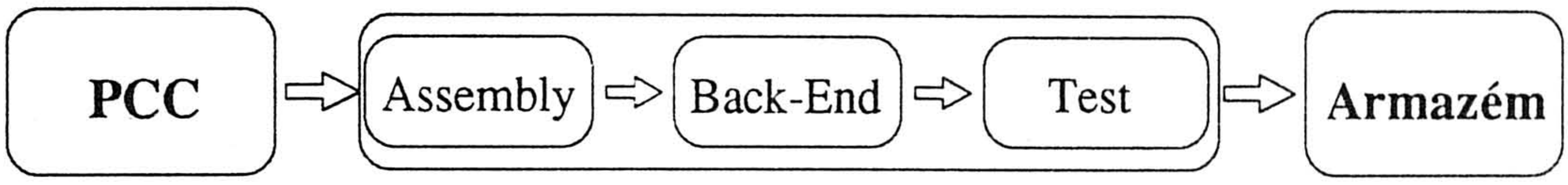
A distribuição normal é útil na medida de distâncias ou tamanhos, como por exemplo o tamanho de um lote. Esse tamanho corresponde a valores que se distribuem em torno de uma dada média.

Os parâmetros de uma distribuição normal são a média (μ) e o desvio padrão (σ). O desvio padrão deve ser superior a 0. A função de probabilidade é:

$$\frac{e^{\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}}{\sigma\sqrt{2\pi}}$$

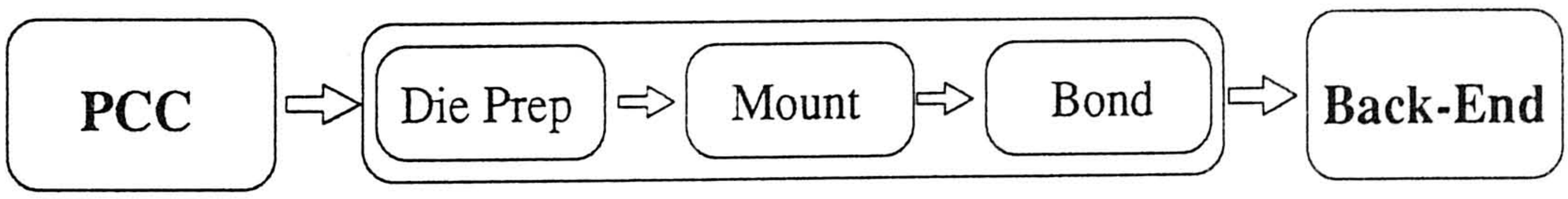
C. Montagem e teste de circuitos integrados

A montagem e teste de circuitos integrados é realizada em três grandes áreas, por sua vez divididas em mais áreas. Dependendo do grau de integração e do tipo de circuito a produzir podem ou não existir diversas áreas de fabrico. O caso a descrever é de média integração e nem todos os tipos de circuito passam no seu percurso de fabrico por todas as áreas. Essas três grandes áreas são denominadas *Assembly*, *Back-end* e *Test*.



C.1 *Assembly*

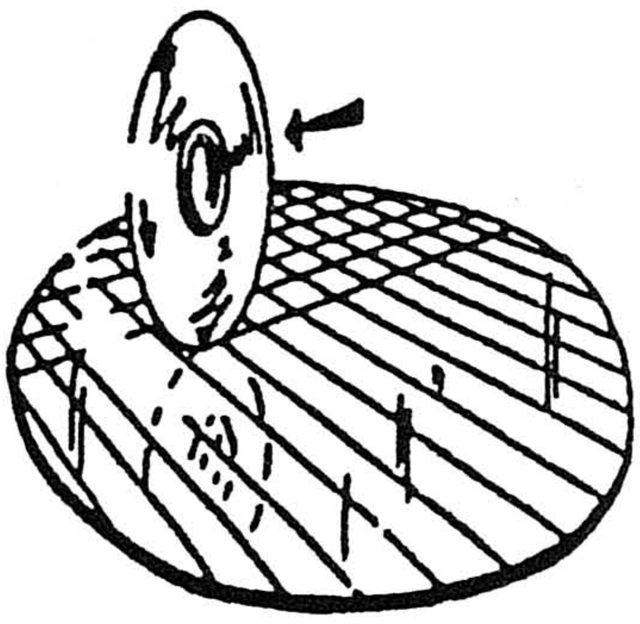
O *assembly* é constituído por três áreas. Essas áreas são denominadas por *Die-prep*, *Mount* e *Bond*. É a área responsável pelo fabrico inicial do circuito. Todas estas áreas têm de se realizar dentro de uma "sala limpa", ou seja uma (ou várias) sala(s) em que o tamanho das partículas que circulam no ar é muito reduzido e permanentemente controlado.

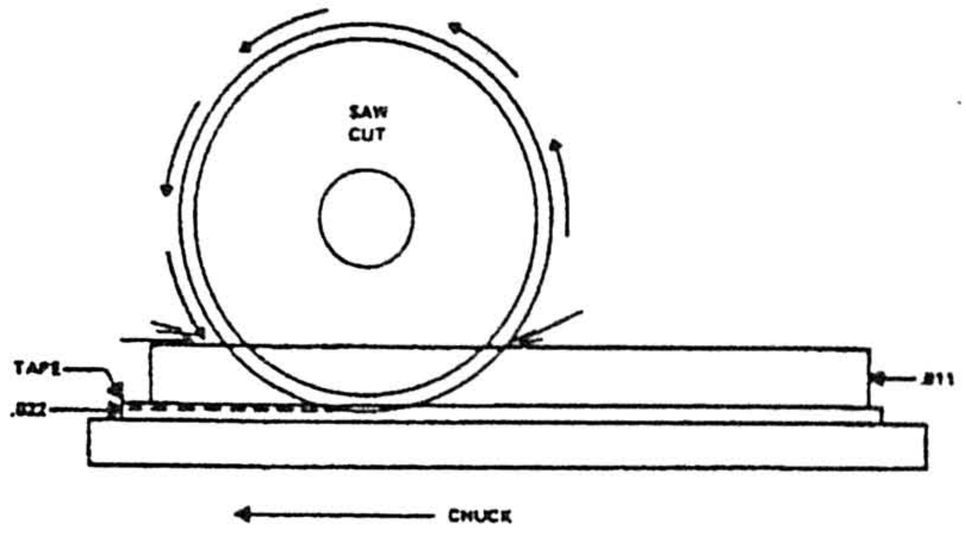


C.1.1 *Die-prep*

A área de *die-prep* é a responsável pela individualização das unidades que são recebidas em *wafers*.

A *wafer* é inicialmente colada numa película que está presa e esticada por um círculo metálico.

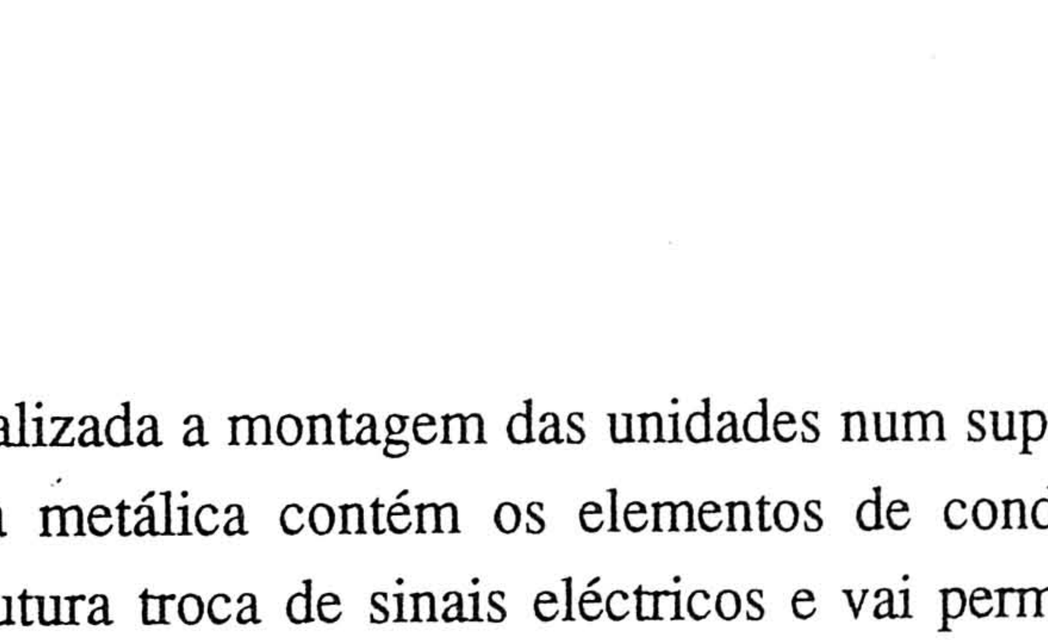




Seguidamente a *wafer* é colocada num equipamento que executa um corte entre as várias unidades, de forma a separá-las. A profundidade de corte é tal que não corta a película a que a *wafer* foi colada. No final desta operação temos as várias centenas ou milhares de *dies* que constituíam a *wafer* individualizados, mas ainda seguros pela película.

C.1.2 *Mount*

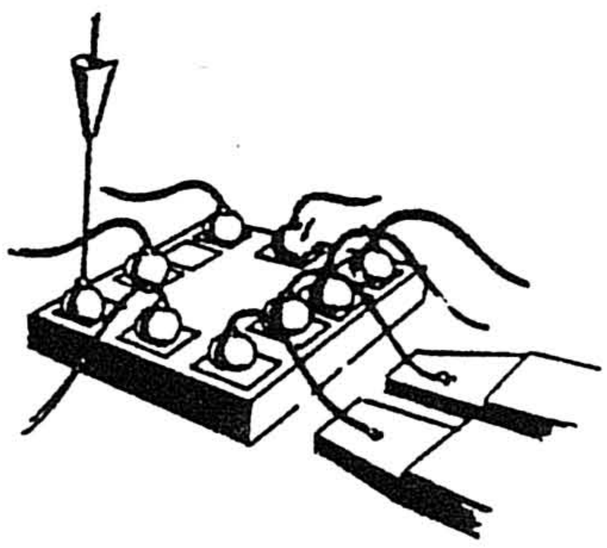
Nesta área é realizada a montagem das unidades num suporte adequado. Esse suporte, de natureza metálica contém os elementos de condutibilidade eléctrica necessários para a futura troca de sinais eléctricos e vai permitir que as unidades sejam agrupadas em pequenos conjuntos facilitando o seu manuseamento. Esse suporte é designado *leadframe* .

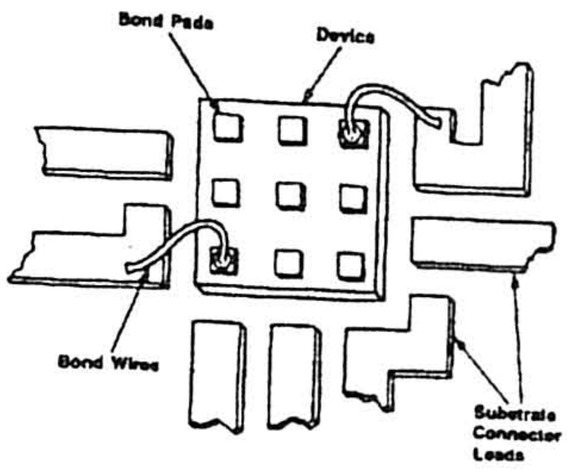


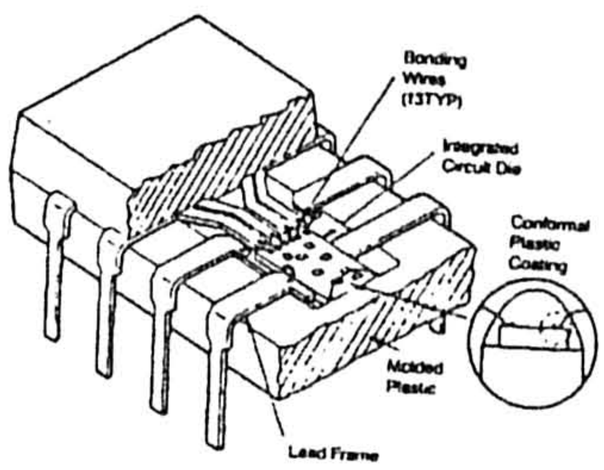
As unidades são retiradas uma a uma da película e coladas ao *leadframe*, orientadas correctamente. Um *leadframe* tem um número variável de unidades, dependendo do tipo de produto que se está a fabricar, mas sempre igual para um mesmo tipo de produto.

C.1.3 *Bond*

Nesta área faz-se a união dos pinos da unidade (*pins*) com os pontos de contacto existentes dentro da unidade em si (*die pads*). Esta união é realizada através de fio de ouro (ou outro metal apropriado). São estas ligações que vão permitir a comunicação entre o exterior e o circuito em si.

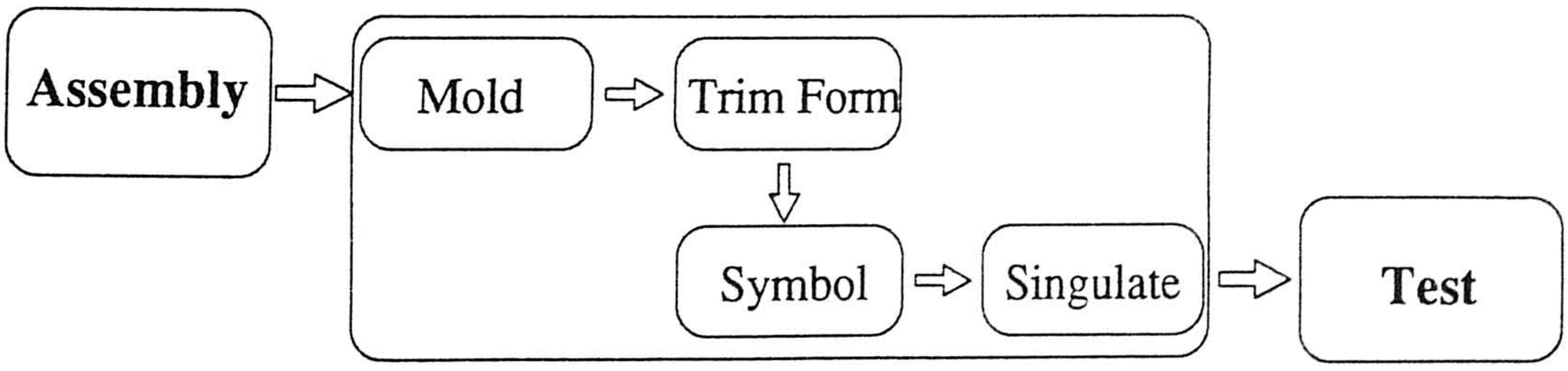






C.2 *Back-end*

O *back-end* é constituído por quatro áreas. Essas áreas são denominadas por *Mold*, *Trim-Form*, *Symbol* e *Singulate*. É a área responsável pelo fabrico final do circuito.

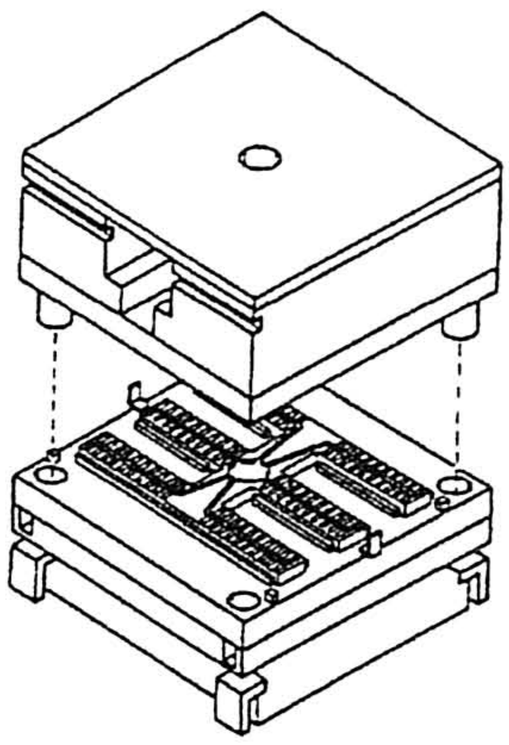


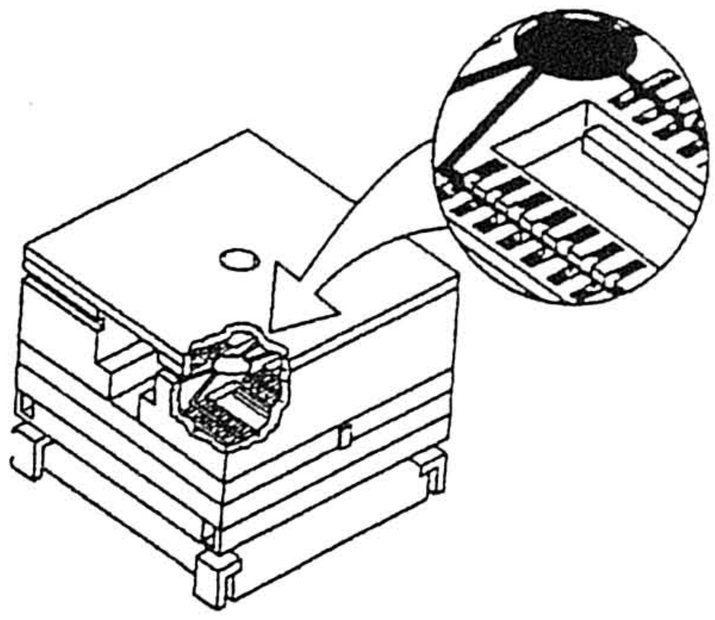
A área de *singulate* é apenas destinada a determinados produtos, como por exemplo SOIC. Os produtos DIP não são processados nesta área.

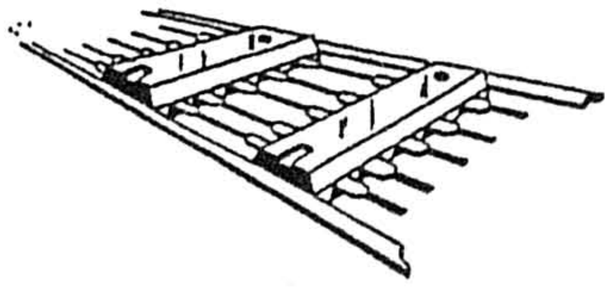
C.2.1 *Mold*

A área de *mold* pode ou não encontrar-se em sala limpa. Em termos de qualidade tal seria desejável, mas as características de trabalho desta área tornam difícil e dispendioso a sua inclusão em sala limpa. A tendência é no entanto para a resolução de tais problemas de forma a que as unidades sejam sempre manuseadas em salas limpas até se encontrarem protegidas do meio ambiente dentro do plástico que as envolverá.

Nesta área os *leadframes* são introduzidos num equipamento de moldagem, o qual injecta um determinada tipo de plástico em volta das unidades. Os *dies* ficam a partir de agora tapados, só sendo possível aceder aos mesmos através dos pinos da unidade.



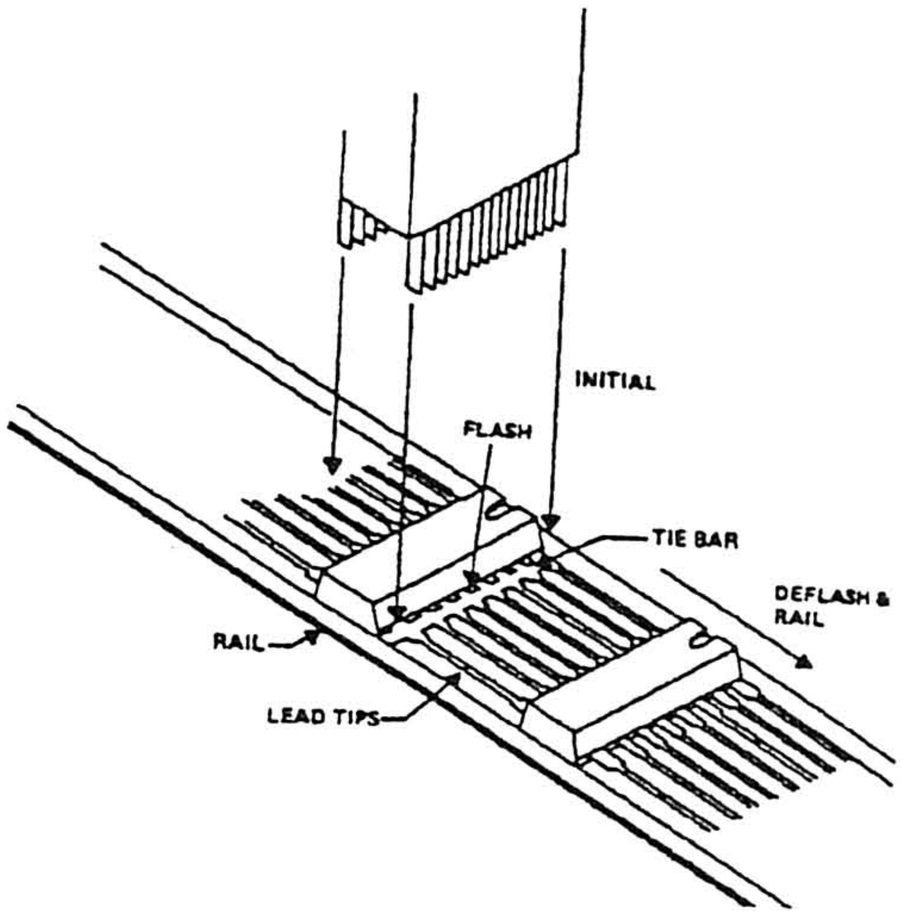


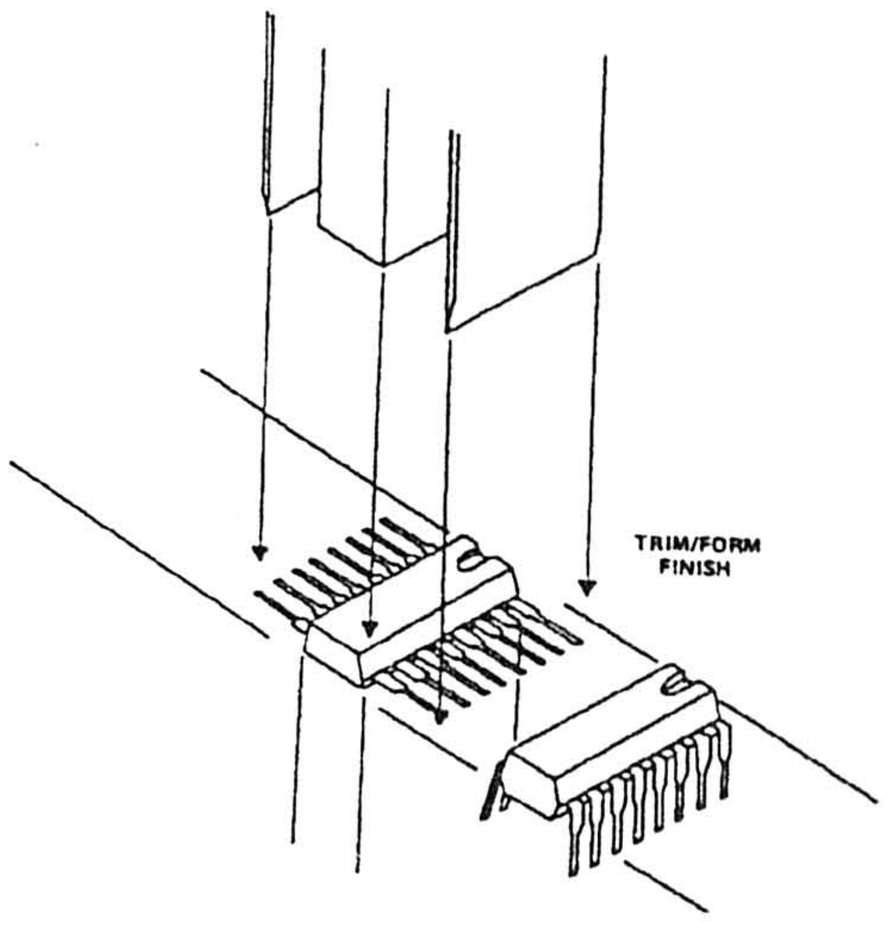


C.2.2 *Trim-Form*

Nesta área é feito o corte (*trim*) dos pinos das unidades relativamente ao leadframe e esses mesmos pinos são dobrados (*form*) de forma adequada.

No caso das unidades DIP, a separação é total, pelo que as unidades passam a estar completamente individualizadas a partir desta área. No caso das unidades SOIC, as unidades continuam presas pelas extremidades ao *leadframe* pois a sua reduzida dimensão tornaria muito difícil o seu manuseamento individualizado na próxima área. A separação completa do *leadframe* será realizada na área de *singulate*.

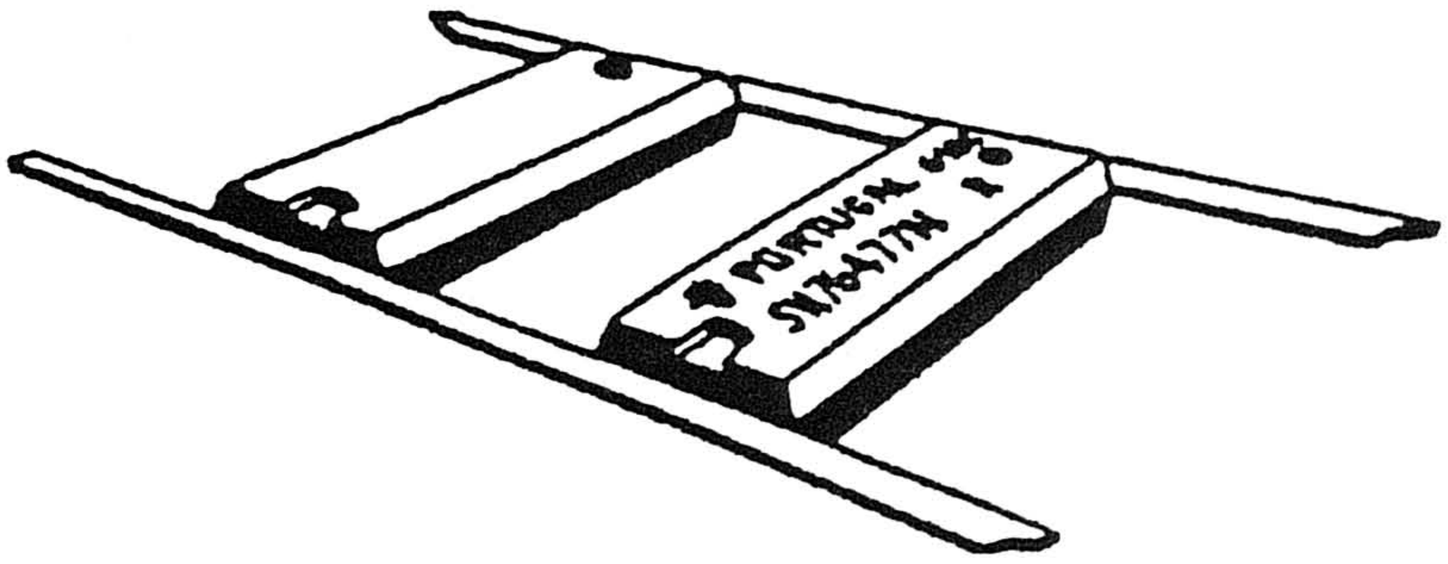




C.2.3 *Symbol*

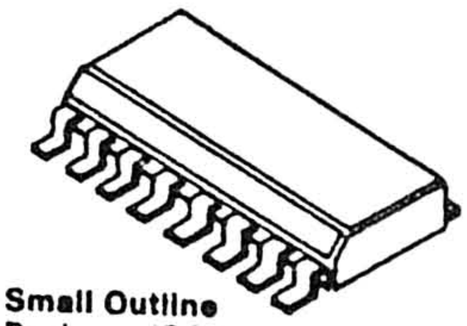
Nesta área é realizada a simbolização das unidades, ou seja a impressão dos caracteres e símbolos que identificam o fabricante e o tipo de unidade.

A simbolização é feita através do recurso a um raio laser que queima a superfície do plástico deixando a impressão feita em baixo relevo. Também pode ser realizada por uma máscara de tinta, mas este processo tende a desaparecer.



C.2.4 *Singulate*

Nesta área (apenas existente para as unidades do tipo SOIC) é feita a separação das unidades do *leadframe* , passando a partir de agora as unidades a estar completamente individualizadas.

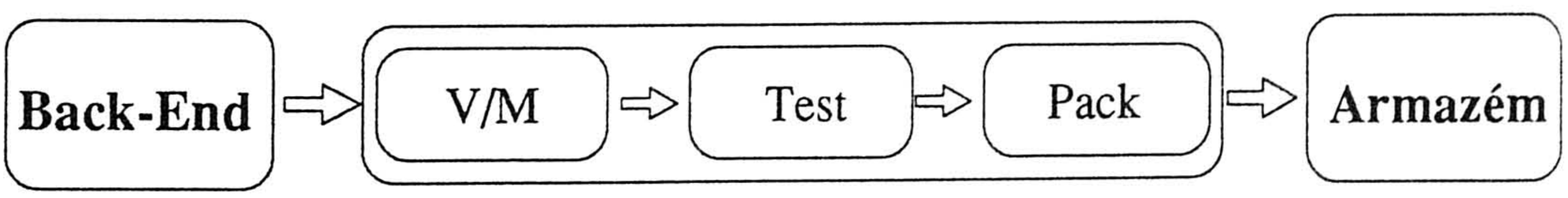


Small Outline
Package (SOP)

C.3 *Test*

O *test* é constituído por três áreas. Essas áreas são denominadas por *V/M*, *Test* e *Pack*. É a área responsável pelo teste final do circuito. As preocupações com a qualidade, embora sempre existentes ao longo de todo o processo de fabrico, tomam aqui um relevo especial, pois é o último filtro existente antes das unidades serem entregues ao cliente.

As unidades antes de estarem individualizadas foram sempre transportadas em suportes metálicos. Depois da sua individualização passam a ser transportadas em tubos plásticos.



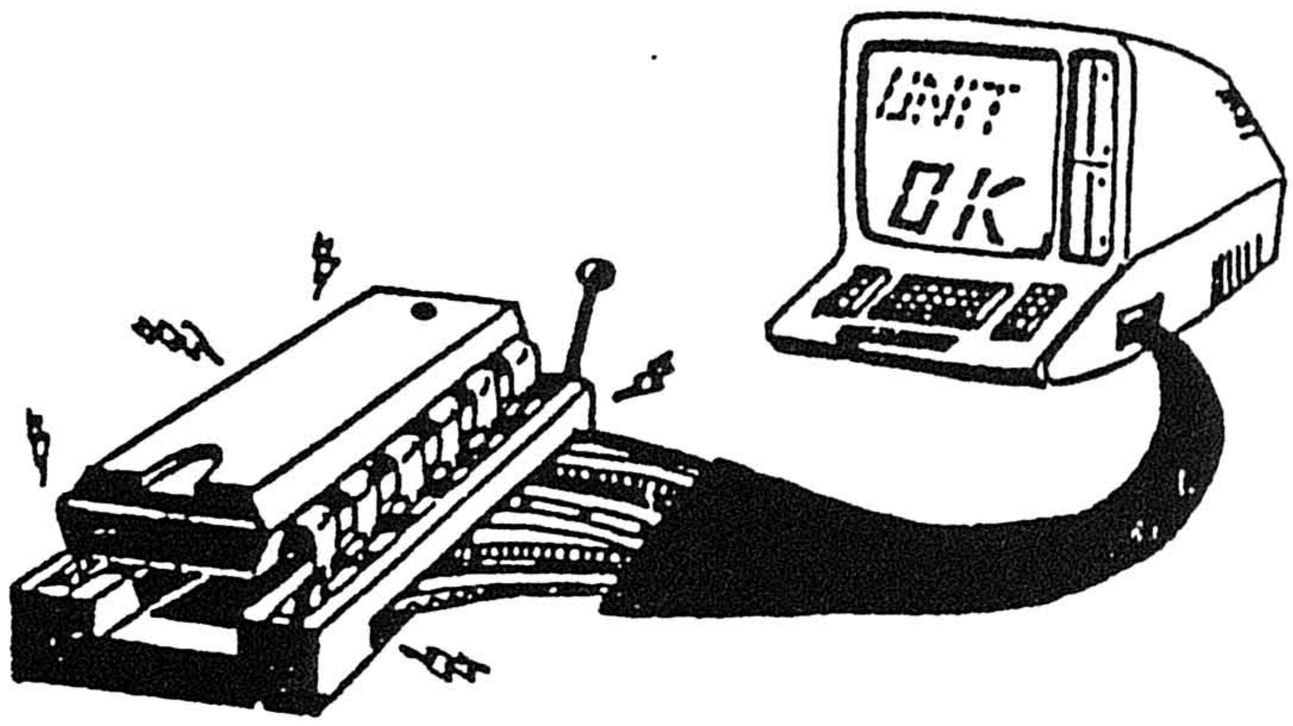
C.3.1 *V/M*

Nesta área as unidades são inspecionadas em termos de simbolização de boa qualidade e não existência de *bent-leads* (pinos torcidos). A designação de V/M vem de teste visual e mecânico.

C.3.2 *Test* propriamente dito

Nesta área é realizado o teste eléctrico às unidades. Dependendo do tipo de unidade a testar, são realizados testes a temperaturas negativas, à temperatura ambiente e a altas temperaturas.

Os testes realizados são de três tipos: de continuidade, testando a ligação de cada pino ao circuito interno; de funcionalidade, testando se o circuito funciona como é esperado; e paramétrico, testando se os parâmetros de funcionamento são os correctos.



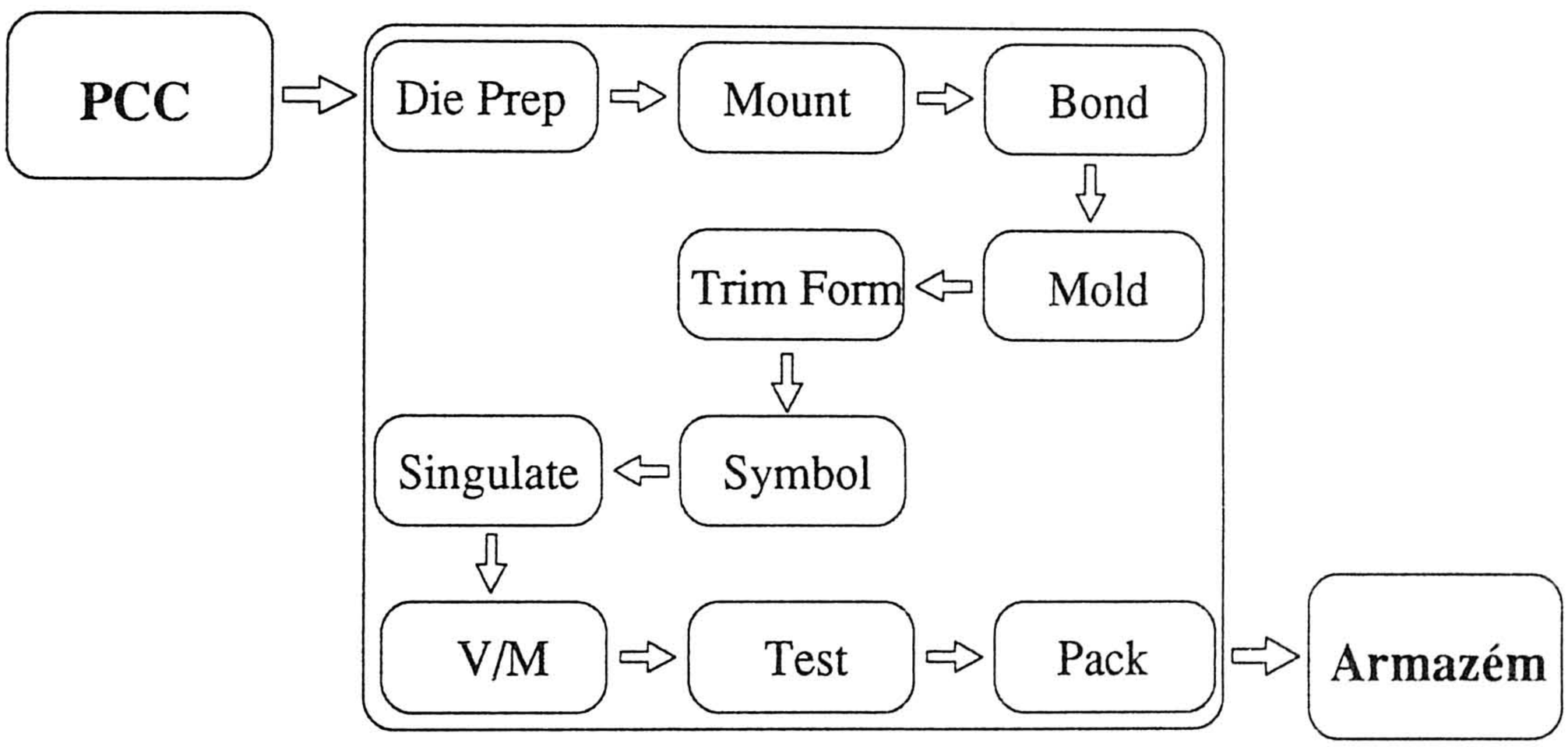
Esta é uma área muito importante devido ao elevado custo dos equipamentos e ao complexo sistema de gestão, pois a variedade de diferentes tipos de teste é muito elevada, variando com o tipo de produto e cliente.

C.3.3 *Pack*

Nesta área é realizado o empacotamento das unidades para serem enviadas para o cliente. Os tubos de plásticos são fechados dentro de caixas. No caso dos SOIC alguns dos lotes vão para um equipamento denominado *tape and reel* onde as unidades são transferidas para um sistema de fita contínua.

Desta área os lotes são enviados para o armazém final onde deixam de estar sob a responsabilidade do FCC.

Apresenta-se de seguida um resumo do processo de fabrico total:



D. Algumas classes da simulação

Nas páginas seguintes apresento algumas das classes que foram definidas para a simulação. O objectivo é essencialmente permitir elucidar alguns dos comentários realizados na parte principal, assim como dar uma ideia da estrutura de uma classe em Smalltalk. Não se apresentam todas as classes do sistema, mas apenas algumas representativas. A maioria das classes aqui apresentadas são baseadas nas recomendadas por Goldberg e Robson.

Índice

Class: **#ProbabilityDistribution** 122
Class: **#ContinuousProbability** 123
Class: **#Exponential** 123
Class: **#Gamma** 124
Class: **#Normal** 125
Class: **#Uniform** 126
Class: **#DiscreteProbability** 127
Class: **#Bernoulli** 127
Class: **#Binomial** 128
Class: **#Geometric** 129
Class: **#Poisson** 130
Class: **#SampleSpace** 131
Class: **#DelayedEvent** 131
Class: **#WaitingSimulationObject** 133
Class: **#Resource** 134
Class: **#ResourceCoordinator** 134
Class: **#ResourceProvider** 136
Class: **#Simulation** 137
Class: **#SimulationObject** 140
Class: **#Histogram** 142
Class: **#StatisticsWithSimulation** 144
Class: **#EventMonitor** 145
Class: **#SimulationObjectRecord** 148

```
Stream subclass: #ProbabilityDistribution
        instanceVariableNames: ''
        classVariableNames: 'U '
        poolDictionaries: ''
        category: 'Statistics'
ProbabilityDistribution comment:
        'Class initialization is needed. Use ProbabilityDistribution initialize'

ProbabilityDistribution methodsFor: 'random sampling'

atEnd
        ^false

next
        "This is a general random number generation method for any probability law;  use the (0,1)
uniformly distributed random varible U as the value of the law's distribution function.  Obtain the
next random value and then solve for the inverse.  The inverse solution is defined by the subclass."
        ^self inverseDistribution: U next

ProbabilityDistribution methodsFor: 'probability functions'

density: x
        self subclassResponsibility

distribution: aCollection
        self subclassResponsibility

ProbabilityDistribution methodsFor: 'private'

computeSample: m outOf: n
        m>n ifTrue: [^0.0].
        ^n factorial / (n-m) factorial

inverseDistribution: x
        self subclassResponsibility


ProbabilityDistribution class
        instanceVariableNames: ''

ProbabilityDistribution class methodsFor: 'instance creation'

new
        ^self basicNew

ProbabilityDistribution class methodsFor: 'class initialization'

initialize
        "Uniformly distributed random numbers in the range [o,1]."
        U _ Random new
```

```
ProbabilityDistribution subclass: #ContinuousProbability
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Statistics'


ContinuousProbability methodsFor: 'probability functions'

distribution: aCollection
        "This is a slow and dirty trapezoidal integration to determine the area under the probability
function curve y=density (x) for x in the specified collection.  The method assumes that the
collection contains numerically-ordered elements."
        | t aStream x1 x2 y1 y2 |
        t _ 0.0.
        aStream _ ReadStream on: aCollection.
        x2 _ aStream next.
        y2 _ self density: x2.
        [x1 _ x2.  x2 _ aStream next]
                whileTrue:
                        [y1 _ y2.
                         y2 _ self density: x2.
                         t _ t + ((x2-x1)*(y2+y1))].
        ^t*0.5


ContinuousProbability subclass: #Exponential
        instanceVariableNames: 'mu '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Statistics'


Exponential methodsFor: 'accessing'

mean
        ^1.0/mu

variance
        ^1.0/(mu*mu)

Exponential methodsFor: 'probability functions'

density: x
        x > 0.0
                ifTrue: [^mu * (mu*x) negated exp]
                ifFalse: [^0.0]

distribution: anInterval
        anInterval last <= 0.0
                ifTrue: [^0.0]
                ifFalse: [^1.0 - (mu * anInterval last) negated exp - (anInterval first > 0.0 ifTrue: [self
distribution: (0.0 to: anInterval first)] ifFalse: [0.0])]
```

```
Exponential methodsFor: 'private'

inverseDistribution: x
        ^ x ln negated / mu

setParameter: p
        mu _ p


Exponential class
        instanceVariableNames: ''

Exponential class methodsFor: 'instance creation'

mean: p
        ^self parameter: 1.0/p

parameter:  p
        p > 0.0
                ifTrue: [^self new setParameter: p]
                ifFalse: [self error: 'The probability parameter must be greater than 0.0']


Exponential subclass. #Gamma
        instanceVariableNames: 'N '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Statistics'


Gamma methodsFor: 'accessing'

mean
        ^super mean*N

variance
        ^super variance*N

Gamma methodsFor: 'probability functions'

density: x
        | t |
        x > 0.0
                ifTrue: [t _ mu * x.
                        ^(mu raisedTo: N) / (self gamma: N) *(x raisedTo: N-1) * t negated exp]
                ifFalse: [^0.0]

Gamma methodsFor: 'private'

gamma: n
        | t |
        t _ n - 1.0.
        ^self computeSample: t outOf: t
```

```
setEvents: events
        N _ events


Gamma class
        instanceVariableNames: ''

Gamma class methodsFor: 'instance creation'

events: k mean: p
        | events |
        events _ k truncated.
        events > 0
                ifTrue: [^(self parameter: events/p) setEvents: events]
                ifFalse: [self error: 'the number of events must be greater than 0']


ContinuousProbability subclass: #Normal
        instanceVariableNames: 'mu sigma '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Statistics'
Normal comment:
'How long before a success occurs or how many events occur in a certain time interval?'

Normal methodsFor: 'accessing'

mean
        ^mu

variance
        ^sigma squared

Normal methodsFor: 'random sampling'

next
        | v1 v2 s rand u |
        rand _ Uniform from: -1.0 to: 1.0.
        [v1 _ rand next.
         v2 _ rand next.
         s _ v1 squared + v2 squared.
         s >= 1] whileTrue.
        u _ (-2.0 * s ln /s) sqrt.
        ^mu + (sigma * v1 *u)

Normal methodsFor: 'probability functions'

density: x
        | twoPi t |
        twoPi _ 2 * 3.1415926536.
        t _ x - mu/sigma.
        ^(-0.5 * t squared) exp / (sigma * twoPi sqrt)
```

```
Normal methodsFor: 'private'

setMean: m standardDeviation: s
        mu _ m.
        sigma _ s
Normal class
        instanceVariableNames: ''

Normal class methodsFor: 'instance creation'

mean: a deviation: b
        b > 0.0
        ifTrue: [^self new setMean: a standardDeviation: b]
        ifFalse: [self error: 'standard deviation must be greater than 0.0']


ContinuousProbability subclass: #Uniform
        instanceVariableNames: 'startNumber stopNumber '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Statistics'


Uniform methodsFor: 'accessing'

mean
        ^ (startNumber + stopNumber)/2

variance
        ^ (stopNumber - stopNumber) squared / 12

Uniform methodsFor: 'probability functions'

density: x
        (x between: startNumber and: stopNumber)
                ifTrue: [^1.0 / (stopNumber - startNumber)]
                ifFalse: [^0]

Uniform methodsFor: 'private'

inverseDistribution: x
        "x is a random  number between 0 and 1"
        ^startNumber + (x * (stopNumber - startNumber))

setStart: begin toEnd: end
        startNumber _ begin.
        stopNumber _ end


Uniform class
        instanceVariableNames: ''
```

```
Uniform class methodsFor: 'instance creation'

from: begin to: end
        begin > end
                ifTrue: [self error: 'illegal interval']
                ifFalse: [^self new setStart: begin toEnd: end]


ProbabilityDistribution subclass: #DiscreteProbability
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Statistics'


DiscreteProbability methodsFor: 'probability functions'

distribution: aCollection
        "Answer the sum of the discrete values of the density function for each element in the
collection."
        | t |
        t _ 0.0.
        aCollection do: [:i | t _ t + (self density: i)].
        ^t


DiscreteProbability subclass: #Bernoulli
        instanceVariableNames: 'prob '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Statistics'
Bernoulli comment:
'Does an event occur? density function answers the probability of occurrence of one of two events'


Bernoulli methodsFor: 'accessing'

mean
        ^prob

variance
        ^prob * (1.0 - prob)

Bernoulli methodsFor: 'probability functions'

density: x
        x=1 ifTrue: [^prob].
        x=0 ifTrue: [^1.0-prob].
        self error: ' outcomes of a Bernoulli can only be 1 or 0'
```

```
Bernoulli methodsFor: 'private'

inverseDistribution: x
        x <= prob
                ifTrue: [^1]
                ifFalse: [^0]

setParameter: aNumber
        prob _aNumber


Bernoulli class
        instanceVariableNames: ''

Bernoulli class methodsFor: 'instance creation'

parameter: aNumber
        (aNumber between: 0.0 and: 1.0)
                ifTrue: [^self new setParameter: aNumber]
                ifFalse: [^self error: 'The probability must be between 0.0 and 1.0']


Bernoulli subclass: #Binomial
        instanceVariableNames: 'N '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Statistics'
Binomial comment:
'How many successes occurred in N trials? The density funciton answers what is the probability
that x successes will occur in the next N trials? i.e., N repeated Bernoulli trials'

Binomial methodsFor: 'random sampling'

next
        |t|
        t _ 0.
        N timesRepeat: [t _ t + super next].
        ^t

Binomial methodsFor: 'probability functions'

density: x
        (x between: 0 and: N)
                ifTrue: [^((self computeSample: x outOf: N) / (self computeSample: x outOf: x)) *
                        (prob raisedTo: x)*((1.0 - prob) raisedTo: N-x)]
                ifFalse: [^0.0]

Binomial methodsFor: 'private'

events: n mean: m
        N _ n truncated.
        self setParameter: m/N
```

```
Binomial class
        instanceVariableNames: ''

Binomial class methodsFor: 'instance creation'

events: n mean: m
        n truncated <= 0 ifTrue: [self error: 'number of events must be > 0'].
        ^self new events: n mean: m
```

```
Bernoulli subclass: #Geometric
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Statistics'
Geometric comment:
'How many repeated, independent Bernoulli trials are needed before the first success is obtained?
e.g., how many seconds before the next car arrives (as versus how many cars arrive in the next 20
sec as in a binomial question)'


Geometric methodsFor: 'accessing'

mean
        ^ 1.0 / prob

variance
        ^ (1.0 - prob) / prob squared

Geometric methodsFor: 'probability functions'

density: x
        x>0 ifTrue: [^prob * ((1.0 - prob) raisedTo: x-1)]
                ifFalse: [^0.0]

Geometric methodsFor: 'private'

inverseDistribution: x
        ^(x ln / (1.0 - prob) ln) ceiling
```

```
Geometric class
        instanceVariableNames: ''

Geometric class methodsFor: 'instance creation'

mean: m
        ^self parameter: m
```

```
DiscreteProbability subclass: #Poisson
        instanceVariableNames: 'mu '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Statistics'
Poisson comment:
'how many events occur in a unit time?
        used for sampling potential demands by customers for service
        The Poisson is typically the rate at which the service is provided.
        density function determines the probability that, in a unit interval, x events will occur.'

Poisson methodsFor: 'accessing'

mean
        ^mu
variance
        ^mu

Poisson methodsFor: 'random sampling'

next
        | p n q |
        p _ mu negated exp.
        n _ 0.
        q _ 1.0.
        [q _ q * U next.
         q >= p]
                whileTrue: [n _ n + 1].
        ^n

Poisson methodsFor: 'probability functions'

density: x
        x >= 0
                ifTrue: [^ ((mu raisedTo: x) * (mu negated exp)) / x factorial]
                ifFalse: [^0.0]

Poisson methodsFor: 'private'

setMean: p
        mu _ p

Poisson class
        instanceVariableNames: ''

Poisson class methodsFor: 'instance creation'

mean: p
        p > 0.0
                ifTrue: [^self new setMean: p]
                ifFalse: [self error: 'mean must be greater than 0.0']
```

```
DiscreteProbability subclass: #SampleSpace
        instanceVariableNames: 'data '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Statistics'


SampleSpace methodsFor: 'probability functions'

density: x
        "x must be in the sample space;  the probability must sum over all occurrences of x in the
sample space."
        (data includes: x)
                ifTrue: [^(data occurrencesOf: x) / data size]
                ifFalse: [^0]


SampleSpace methodsFor: 'private'

inverseDistribution: x

        ^data at: (x*data size) truncated + 1

setData: aCollection

        data _ aCollection


SampleSpace class
        instanceVariableNames: ''

SampleSpace class methodsFor: 'instance creation'

data: aCollection

        ^self new setData: aCollection


Object subclass: #DelayedEvent
        instanceVariableNames: 'resumptionSemaphore resumptionCondition '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulations'


DelayedEvent methodsFor: 'accessing'
```

```
condition
	"Answer a condition under which the event should be sequenced."
	^resumptionCondition

condition: anObject
	"The argument, anObject, is the condition under which the event should be sequenced."
	resumptionCondition _ anObject

DelayedEvent methodsFor: 'comparing'

<= aDelayedEvent
	"Answer whether the receiver should be sequenced before the argument."
	resumptionCondition isNil
		ifTrue: [^true]
		ifFalse: [^resumptionCondition <= aDelayedEvent condition]

DelayedEvent methodsFor: 'scheduling'

pause
	"Suspend the current active process, that is, the current event that is running."
	Simulation active stopProcess.
	resumptionSemaphore wait

resume
	"Resume the suspended process."
	Simulation active startProcess.
	resumptionSemaphore signal.
	^resumptionCondition

DelayedEvent methodsFor: 'private'

initialize
	resumptionSemaphore _ Semaphore new

setCondition: anObject
	self initialize.
	resumptionCondition _ anObject


DelayedEvent class
	instanceVariableNames: ''

DelayedEvent class methodsFor: 'instance creation'

new
	^super new initialize

onCondition: anObject
	^super new setCondition: anObject
```

```
DelayedEvent subclass: #WaitingSimulationObject
        instanceVariableNames: 'amount resource '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulations'


WaitingSimulationObject methodsFor: 'task language'

consume: aNumber
        amount _ (amount - aNumber) max: 0

release
        resource produce: amount.
        amount _ 0

release: anAmount
        resource produce: anAmount.
        amount _ amount - anAmount

WaitingSimulationObject methodsFor: 'accessing'

amount
        ^amount

name
        ^resource name

resource
        ^resource

resource: aResource
        resource _ aResource

WaitingSimulationObject methodsFor: 'private'

setAmount: aNumber resource: aResource
        amount _ aNumber.
        resource _ aResource


WaitingSimulationObject class
        instanceVariableNames: ''

WaitingSimulationObject class methodsFor: 'instance creation'

for: amount of: aResource withPriority: aNumber
        ^(self onCondition: aNumber) setAmount: amount resource: aResource
for: amount withPriority: aNumber
        ^(self onCondition: aNumber) setAmount: amount resource: nil
```

```
Object subclass: #Resource
        instanceVariableNames: 'pending resourceName '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulations'


Resource methodsFor: 'accessing'

acquire
        ^self

addRequest: aDelayedEvent
        pending add: aDelayedEvent.
        self provideResources.
        aDelayedEvent pause.

name
        ^resourceName

Resource methodsFor: 'private'

provideResources
        ^self

setName: aString
        resourceName _ aString.
        pending _ SortedCollection new


Resource class
        instanceVariableNames: ''

Resource class methodsFor: 'instance creation'

named: resourceName
        ^self new setName: resourceName



Resource subclass: #ResourceCoordinator
        instanceVariableNames: 'whoIsWaiting '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulations'


ResourceCoordinator methodsFor: 'accessing'

customersWaiting
        ^whoIsWaiting == #customer
```

```
queueLength
        ^pending size

serversWaiting
        ^whoIsWaiting == #server

ResourceCoordinator methodsFor: 'task language'

acquire
        |waiting|
        self customersWaiting ifTrue: [^self giveService].
        "get here if there is no customer waiting for the server."
        waiting _ WaitingSimulationObject for: 1 withPriority: 0.
        whoIsWaiting _ #server.
        self addRequest: waiting.
        ^waiting resource

producedBy: aCustomer
        |waiting|
        waiting _ WaitingSimulationObject for: 1
                                        of: aCustomer
                                        withPriority: 0.
        self serversWaiting ifTrue: [^self getServiceFor: waiting].
        whoIsWaiting _ #customer.
        self addRequest: waiting

ResourceCoordinator methodsFor: 'private'

getServiceFor: aCustomerRequest
        | aServerRequest |
        aServerRequest _ pending removeFirst.
        pending isEmpty ifTrue: [whoIsWaiting _ #none].
        aServerRequest resource: aCustomerRequest.
        aServerRequest resume.
        aCustomerRequest pause

giveService
        |aCustomerRequest|
        aCustomerRequest_ pending removeFirst.
        pending isEmpty ifTrue: [whoIsWaiting _ #none].
        ^aCustomerRequest

setName: aString
        super setName: aString.
        whoIsWaiting _ #none
```

```
Resource subclass: #ResourceProvider
        instanceVariableNames: 'amountAvailable '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulations'


ResourceProvider methodsFor: 'accessing'

amountAvailable
        ^amountAvailable

ResourceProvider methodsFor: 'task language'

acquire: amountNeeded withPriority: priorityNumber
        | waiting |
        waiting _ WaitingSimulationObject
                                        for: amountNeeded
                                        of: self
                                        withPriority: priorityNumber.
        self addRequest: waiting.
        ^waiting

produce: amount
        amountAvailable _ amountAvailable + amount.
        self provideResources

ResourceProvider methodsFor: 'private'

provideResources
        | waiting |
        [pending isEmpty not and: [pending first amount <= amountAvailable]]
                whileTrue:
                        [waiting _ pending removeFirst.
                        amountAvailable _ amountAvailable - waiting amount.
                        waiting resume]

setName: aResourceName with: amount
        super setName: aResourceName.
        amountAvailable _ amount

ResourceProvider class
        instanceVariableNames: ''

ResourceProvider class methodsFor: 'instance creation'

named: aResourceName
        ^self new setName: aResourceName with: 0

named: aResourceName with: amount
        ^self new setName: aResourceName with: amount
```

```
Object subclass: #Simulation
        instanceVariableNames: 'resources currentTime eventQueue processCount '
        classVariableNames: 'RunningSimulation '
        poolDictionaries: ''
        category: 'Simulations'
Simulation comment:
'Expects to handle several kinds of resources, represented abstractly by class Resource, and
concretely by subclasses of Resource:  ResourceProvider, StaticResource, and
ResourceCoordinator'

Simulation methodsFor: 'initialization'

activate
        "This instance is now the active simulation"
        RunningSimulation _ self

defineArrivalSchedule
        "A subclass specifies the schedule by which simulation objects dynamically enter into the
simulation."
        ^self

defineResources
        "A subclass specifies the schedule by which simulation objects that are initially entered into
the simulation."
        ^self

initialize
        resources _ Set new.
        currentTime _ 0.0.
        processCount _ 0.
        eventQueue _ SortedCollection new

Simulation methodsFor: 'task language'

coordinate: resourceName
        (self includesResourceFor: resourceName)
                ifFalse: [resources add: ( ResourceCoordinator named: resourceName)]

produce: amount of: resourceName
        (self includesResourceFor: resourceName)
                ifTrue: [(self provideResourceFor: resourceName)
                                produce: amount
                        "sends produce: to a ResourceProvider"]
                ifFalse: [resources add: (ResourceProvider named: resourceName with: amount)]

schedule: actionBlock after: timeDelay
        self schedule: actionBlock at: currentTime + timeDelay

schedule: aBlock at: timeInteger
        "This is the mechanism for scheduling a single action."
        self newProcessFor: [self delayUntil: timeInteger.  aBlock value]
```

```
scheduleArrivalOf: aSimulationObjectClass accordingTo: aProbabilityDistribution
        "This means start now"
        self
                scheduleArrivalOf: aSimulationObjectClass
                accordingTo: aProbabilityDistribution
                startingAt: currentTime

scheduleArrivalOf: aSimulationObjectClass accordingTo: aProbabilityDistribution startingAt:
timeInteger
        "Note that aSimulationObjectClass is the class Simulation or one of
        its subclasses.  The real work is done in the private message
        schedule:startingAt:andThenEvery:. "
        self
                schedule: [aSimulationObjectClass new startUp]
                startingAt: timeInteger
                andThenEvery: aProbabilityDistribution

scheduleArrivalOf: aSimulationObject at: timeInteger
        self schedule: [aSimulationObject startUp]
                at: timeInteger

Simulation methodsFor: 'scheduling'

delayFor: timeDelay
        self delayUntil: currentTime+timeDelay

delayUntil: aTime
        | delayEvent |
        delayEvent _ DelayedEvent onCondition: aTime.
        eventQueue add: delayEvent.
        "self stopProcess."
        delayEvent pause.
        "self startProcess"

newProcessFor: aBlock
        self startProcess.
        [aBlock value. self stopProcess] fork

startProcess
        processCount _ processCount +1

stopProcess
        processCount _ processCount - 1

Simulation methodsFor: 'simulation control'

enter: anObject
        ^self

exit: anObject
        ^self
```

```
finishUp
	"We need to empty out the event queue."
	eventQueue _ SortedCollection new.
	^nil

proceed
	| eventProcess |
	[self readyToContinue]
		whileFalse: [Processor yield].
	eventQueue isEmpty
		ifTrue: [^self finishUp]
		ifFalse:
			[eventProcess _ eventQueue removeFirst.
			currentTime _ eventProcess condition.
			eventProcess resume]

startUp
	self activate.
	self defineResources.
	self defineArrivalSchedule

Simulation methodsFor: 'accessing'

includesResourceFor: resourceName
	| test |
	test _ resources
			detect: [:each | each name = resourceName]
			ifNone: [nil].
	^test notNil

provideResourceFor: resourceName
	^resources detect: [ :each | each name = resourceName]

time
	^currentTime

Simulation methodsFor: 'private'

readyToContinue
	^processCount = 0

schedule: aBlock startingAt: timeInteger andThenEvery: aProbabilityDistribution
	self newProcessFor:
			[self delayUntil: timeInteger.
			self newProcessFor: aBlock copy.
			aProbabilityDistribution do:
				[:nextTimeDelay |
				self delayFor: nextTimeDelay.
				self newProcessFor: aBlock copy]]

Simulation methodsFor: 'reporting'
```

```
report: label
        Transcript show: label.
        Transcript tab.
        Transcript show: 'pc: ', processCount printString.
        Transcript space.
        Transcript show: 'procs: ', (Processor processesAt: Processor userSchedulingPriority)
printString.
        Transcript space.
        Transcript show: 'events: ', eventQueue size printString.
        Transcript cr

Simulation class
        instanceVariableNames: ''

Simulation class methodsFor: 'instance creation'

new
        ^super new initialize

Simulation class methodsFor: 'accessing'

active
        ^RunningSimulation

Object subclass: #SimulationObject
        instanceVariableNames: ''
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulations'
SimulationObject comment:
'A SimulationObject represents any object that can be given a sequence of tasks to do.  The class
specifies a general control sequence by which the object enters, carries out its tasks, and leaves
the simulation. Resources can be produced, consumed, acquired, and tested by queries such as
amountAvailable, serversWaiting, custoemrsWaiting?'

SimulationObject methodsFor: 'initialization'

initialize
        "Do nothing.  Subclasses will initialize instance variables."
        ^self

SimulationObject methodsFor: 'simulation control'

finishUp
        "Tell the simulation that the receiver is done with its tasks."
        Simulation active exit: self
```

```
startUp
        Simulation active enter: self.
        "First tell the simulation that the receiver is beginning to do my tasks."
        self tasks.
        self finishUp

tasks
        "Do nothing.  Subclasses will schedule activities."
        ^self

SimulationObject methodsFor: 'task language'

acquire: amount ofResource: resourceName
        "Get the resource and then tell it to acquire amount of it.  Answers
        an instance of StaticResource."
        ^(Simulation active provideResourceFor: resourceName)
                acquire: amount withPriority: 0

acquire: amount ofResource: resourceName withPriority: priority
        "Returns a StaticResource"
        ^(Simulation active provideResourceFor: resourceName)
                acquire: amount withPriority: priority

acquireResource: resourceName
        "Returns a StaticResource"
        ^(Simulation active provideResourceFor: resourceName) acquire

holdFor: aTimeDelay
        Simulation active delayFor: aTimeDelay

inquireFor: amount ofResource: resourceName
        ^(Simulation active provideResourceFor: resourceName) amountAvailable >= amount

numberOfProvidersOfResource: resourceName
        | resource |
        resource _ Simulation active provideResourceFor: resourceName.
        resource serversWaiting
                ifTrue: [^resource queueLength]
                ifFalse: [^0]

numberOfRequestersOfResource: resourceName
        | resource |
        resource _ Simulation active provideResourceFor: resourceName.
        resource customersWaiting
                ifTrue: [^resource queueLength]
                ifFalse: [^0]

produce: amount ofResource: resourceName
        Simulation active produce: amount of: resourceName

produceResource: resourceName
        ^(Simulation active provideResourceFor: resourceName) producedBy: self
```

```
release: aStaticResource
        ^aStaticResource release

resourceAvailable: resourceName
        "Does the active simulaton have a resource with this attribute available?"
        ^Simulation active includesResourceFor: resourceName

resume: anEvent
        ^anEvent resume
stopSimulation
        Simulation active finishUp
```

```
SimulationObject class
        instanceVariableNames: ''

SimulationObject class methodsFor: 'instance creation'

new
        ^super new initialize
```

```
Object subclass: #Histogram
        instanceVariableNames: 'tallyArray lowerBound upperBound step minValue maxValue
totalValues extraEntries '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulation statistics gathering'

Histogram methodsFor: 'accessing'

contains: aValue
        ^lowerBound <= aValue and: [aValue < upperBound]

store: aValue
        | index |
        minValue isNil
                ifTrue: [minValue _ maxValue _ aValue]
                ifFalse:
                        [minValue _ minValue min: aValue.
                        maxValue _ maxValue max: aValue].
        totalValues _ totalValues + aValue.
        (self contains: aValue)
                ifTrue:
                        [index _ (aValue - lowerBound // step) + 1.
                        tallyArray at: index put: (tallyArray at: index) + 1]
                ifFalse: [extraEntries _ extraEntries + 1]

Histogram methodsFor: 'printing'
```

```
firstHeader: aStream
        aStream cr; tab.
        aStream nextPutAll: 'Number of '.
        aStream tab.
        aStream nextPutAll: 'Minimum '.
        aStream tab.
        aStream nextPutAll: 'Maximum '.
        aStream tab.
        aStream nextPutAll: 'Average '.
        aStream cr; tab.
        aStream nextPutAll: 'Objects '.
        aStream tab.
        aStream nextPutAll: 'Value '.
        aStream tab.
        aStream nextPutAll: 'Value '.
        aStream tab.
        aStream nextPutAll: 'Value '.

printStatisticsOn: aStream
        | totalObjs pos |
        self firstHeader: aStream.
        aStream cr; tab.
        totalObjs _ extraEntries.
        tallyArray do: [:each | totalObjs _ totalObjs + each].
        totalObjs printOn: aStream.
        aStream tab.
        minValue printOn: aStream.
        aStream tab.
        maxValue printOn: aStream.
        aStream tab.
        (totalValues / totalObjs) asFloat printOn: aStream.
        aStream cr.
        self secondHeader: aStream.
        aStream cr.
        pos _ lowerBound.
        tallyArray do:
                [:entry |
                pos printOn: aStream.
                aStream nextPut: $-.
                (pos _ pos + step) printOn: aStream.
                aStream tab.
                entry printOn: aStream.
                aStream tab.
                (entry / totalObjs) asFloat printOn: aStream.
                aStream tab.
                aStream nextPut: $|.
                entry rounded timesRepeat: [aStream nextPut: $X].
                aStream cr]
```

```
secondHeader: aStream
        aStream cr; tab.
        aStream nextPutAll: 'Number of '.
        aStream cr.
        aStream nextPutAll: 'Entry '.
        aStream tab.
        aStream nextPutAll: 'Objects '.
        aStream tab.
        aStream nextPutAll: 'Frequency '.

Histogram methodsFor: 'private'

newLower: lowerNum upper: upperNum by: stepAmount
        tallyArray _ Array new: (upperNum - lowerNum // stepAmount).
        tallyArray atAllPut: 0.
        lowerBound _ lowerNum.
        upperBound _ upperNum.
        step _ stepAmount.
        minValue _ maxValue _ nil.
        totalValues _ 0.
        extraEntries _ 0


Histogram class
        instanceVariableNames: ''

Histogram class methodsFor: 'class initialization'

from: lowerNum to: upperNum by: step
        ^self new newLower: lowerNum upper: upperNum by: step


Simulation subclass: #StatisticsWithSimulation
        instanceVariableNames: 'statistics '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulation statistics gathering'

StatisticsWithSimulation methodsFor: 'initialization'

initialize
        super initialize.
        statistics _ Dictionary new.

StatisticsWithSimulation methodsFor: 'simulation scheduling'

enter: anObject
        statistics at: anObject put: (SimulationObjectRecord new entrance: currentTime)

exit: anObject
        (statistics at: anObject) exit: currentTime
```

```
StatisticsWithSimulation methodsFor: 'statistics'

printStatisticsOn: aStream
        | stat |
        aStream cr.
        aStream nextPutAll: ' Object'.
        aStream tab.
        aStream nextPutAll: 'Entrance Time'.
        aStream tab.
        aStream nextPutAll: 'Duration'.
        aStream cr.
        stat _ SortedCollection sortBlock: [:i :j | i value entrance <= j value entrance].
        statistics associationsDo: [:each | stat add: each].
        stat do:
                [:anAssociation |
                aStream cr.
                anAssociation key printOn: aStream.
                aStream tab.
                anAssociation value printOn: aStream]


SimulationObject subclass: #EventMonitor
        instanceVariableNames: 'label '
        classVariableNames: 'Counter DataFile '
        poolDictionaries: ''
        category: 'Simulation statistics gathering'


EventMonitor methodsFor: 'scheduling'

finishUp
        super finishUp.
        self timeStamp.
        DataFile nextPutAll: ' exits '.

startUp
        self timeStamp.
        DataFile nextPutAll: ' enters '.
        super startUp

EventMonitor methodsFor: 'task language'

acquire: amount ofResource: resourceName
        | aStaticResource |
        self timeStamp.
        DataFile nextPutAll: ' requests '.
        amount printOn: DataFile.
        DataFile nextPutAll: ' of ', resourceName.
        aStaticResource _ super acquire: amount ofResource: resourceName.
        self timeStamp.
        DataFile nextPutAll: ' obtained '.
```

```
        amount printOn: DataFile.
        DataFile nextPutAll: ' of ', resourceName.
        ^aStaticResource

acquire: amount ofResource: resourceName withPriority: priorityNumber
        | aStaticResource |
        self timeStamp.
        DataFile nextPutAll: ' requests '.
        amount printOn: DataFile.
        DataFile nextPutAll: ' at priority '.
        priorityNumber printOn: DataFile.
        DataFile nextPutAll: ' of ', resourceName.
        aStaticResource _ super acquire: amount ofResource: resourceName withPriority:
priorityNumber.
        self timeStamp.
        DataFile nextPutAll: ' obtained '.
        amount printOn: DataFile.
        DataFile nextPutAll: ' of ', resourceName.
        ^aStaticResource

acquireResource: resourceName
        | anEvent |
        self timeStamp
        DataFile nextPutAll: ' wants to serve for '.
        DataFile nextPutAll: resourceName.
        anEvent _ super acquireResource: resourceName.
        self timeStamp.
        DataFile nextPutAll: ' can serve '.
        anEvent resource printOn: DataFile.
        ^anEvent

holdFor: aTimeDelay
        self timeStamp.
        DataFile nextPutAll: ' holds for '.
        aTimeDelay printOn: DataFile.
        super holdFor: aTimeDelay

produce: amount ofResource: resourceName
        self timeStamp.
        DataFile nextPutAll: ' produces '.
        amount printOn: DataFile.
        DataFile nextPutAll: ' of ', resourceName.
        super produce: amount ofResource: resourceName

produceResource: resourceName
        self timeStamp.
        DataFile nextPutAll: ' wants to get service as '.
        DataFile nextPutAll: resourceName.
        super produceResource: resourceName
```

```
release: aStaticResource
        self timeStamp.
        DataFile nextPutAll: ' releases '.
        aStaticResource amount printOn: DataFile.
        DataFile nextPutAll: ' of ', aStaticResource name.
        super release: aStaticResource

resume: anEvent
        self timeStamp.
        DataFile nextPutAll: ' resumes '.
        anEvent resource printOn: DataFile.
        super resume: anEvent

EventMonitor methodsFor: 'private'

timeStamp
        DataFile cr.
        Simulation active time printOn: DataFile.
        DataFile tab.
        self printOn: DataFile

EventMonitor methodsFor: 'initialization'

initialize
        super initialize.
        self setLabel

EventMonitor methodsFor: 'accessing'

label
        ^label

setLabel
        Counter _ Counter + 1.
        label _ Counter printString.

EventMonitor methodsFor: 'printing'

printOn: aStream
        self class name printOn: aStream.
        aStream space.
        aStream nextPutAll: self label


EventMonitor class
        instanceVariableNames: ''

EventMonitor class methodsFor: 'class initialization'

file: aFile
        DataFile _ aFile.
        Counter _ 0
```

```
Object subclass: #SimulationObjectRecord
        instanceVariableNames: 'entranceTime duration '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Simulation statistics gathering'


SimulationObjectRecord methodsFor: 'accessing'

duration
        ^duration

entrance
        ^entranceTime

entrance: currentTime
        entranceTime _ currentTime

exit
        ^entranceTime + duration

exit: currentTime
        duration _ currentTime - entranceTime

SimulationObjectRecord methodsFor: 'printing'

printOn: aStream
        entranceTime printOn: aStream.
        aStream tab.
        duration printOn: aStream
```


BIBLIOTECA F.E.

FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO
BIBLIOTECA
0000027073