Software Design Baseado em Encapsulamento de Dados e Troca de Mensagens entre Subsistemas Autônomos — Focalizando o Reúso —

Maria Luiza d'Almeida Sanchez
UFF (TET/CAA) - Departamento de Engenharia de Telecomunicações
Pós -Graduação em Computação Aplicada e Automação

e-mail: mluiza@caa.uff.br

Bruno Maffeo[1]
DEI - Departamento de Estatística e Informática
IMF - Instituto de Matemática e Física
UFG - Universidade Federal de Goiás

e-mail: bmaffeo@dei.ufg.br

PUC-Rio Inf MCC17/96 Junho, 1996

Abstract

This work presents a method of software design for real-time control systems. The method is based on "Information Hiding and Exchange of Messages between Independent Subsystems". Employing the data encapsulation concept and concepts inherited from development techniques using Configuration Languages, the method partitions a complex system into more manageable subsystems intended for reuse in other systems that have to perform similar functions.

Keywords:
Design Method, Real-Time Systems, Reuse, Information Hiding, Configuration Languages

Resumo

Este trabalho apresenta um método de design para o software de sistemas de tempo-real para controle de processos. O método baseia-se em "Encapsulamento de Dados e Troca de Mensagens entre Subsistemas Autônomos". Empregando o conceito de encapsulamento de dados e conceitos herdados de técnicas de desenvolvimento que usam Linguagens de Configuração, o método particiona um sistema complexo em subsistemas mais simples projetados para reúso em outros sistemas que devam executar funções similares.

Palavras Chaves:
Método de Design, Sistemas de Tempo-Real, Reúso, Encapsulamento de Dados, Linguagens de Configuração

---

[1] professor visitante sob licença do Departamento de Informática da PUC-Rio.

Sumário

1. Introdução

Entre a representação conceitual de um sistema de software e sua implementação existe um desnível muito grande de abstração bem como acentuadas distinções de complexidade [McMenamin84, Ward85, Maffeo92]. O preenchimento da lacuna correspondente constitui, normalmente, uma das etapas mais demoradas do processo de desenvolvimento de sistemas e emprega quantidade significativa de recursos materiais e humanos. Vários esforços vêm sendo realizados [Rumbaugh91, Bustard88, Ward85, Jackson83, Booch94] visando cumprir adequadamente essa etapa e, em conseqüência, reduzir custo, prazo e risco associados a projetos de desenvolvimento de sistemas.

A motivação deste trabalho origina-se nesse tipo de questão e seu foco direciona-se para conceitos, ferramentas e técnicas que visam à implantação do reúso, em especial no caso de sistemas de tempo-real para controle de processos, de grande porte, em ambiente distribuído. Devido à complexidade dessa classe de sistemas, cada projeto constitui investimento de risco associado a uma probabilidade elevada de não haver sucesso no desenvolvimento e, assim, de inexistir retorno sobre o investimento realizado.

Nessas condições, é fundamental empregar ferramentas conceituais e técnicas de desenvolvimento que:

- permitam a redução:
  - dos custos e do risco durante a construção do sistema;
  - do custo de sua manutenção corretiva e/ou evolutiva;
- apóiem o estudo da concorrência inerente ao software de controle;
- aumentem a portatilidade desse software para diversas arquiteturas de hardware.

O estabelecimento dessas condições poderia viabilizar projetos importantes que hoje não são executados pois o cliente/usuário não se dispõe a enfrentar todos os ônus que podem decorrer. A ênfase no reúso de subsistemas que compõem o software de controle foi determinante para a escolha da estrutura proposta para o processo de desenvolvimento e, em particular, para a definição do método de design.

A introdução de alterações durante e após o desenvolvimento de um sistema é inevitável. A maior dificuldade para introduzir essas alterações reside na superação de decisões rígidas de design. Segundo [Veldwijk94], maior flexibilidade de design é a chave para enfrentar a forma atual pela qual manifesta-se a "crise do software" — basicamente associada ao elevado custo de manutenção do sistema construído —. O design de um sistema será tanto mais flexível quanto mais independentes entre si forem os componentes do sistema resultante do processo de desenvolvimento. Essa independência pode ser parcialmente baseada no encapsulamento de informação [Parnas72], o que aproxima a abordagem proposta neste trabalho de uma abordagem orientada a objetos, e finalmente obtida pelo design independente dos subsistemas que compõem a configuração do software. O emprego dessa estratégia de design dá origem a processos de construção e de manutenção baseados em software altamente reusável.

A utilização de linguagens de configuração, inicialmente proposta por [DeRemer76], é sugerida em diversos contextos, tais como: MESA [Lauer79], CONIC [Sloman85], MASCOT [Simpson86] e MILs [Prieto-Diaz86]. Essas linguagens permitem a definição de cada componente independentemente de sua interconexão com outros componentes. A atividade de produzir um componente é conhecida como programação em ponto pequeno ("programming-in-small") e a definição da interconexão de componentes como programação em ponto grande ("programming-in-large"). Esses trabalhos confirmam os benefícios de tratar separadamente:

- a especificação dos componentes de software, que implementam as funções do sistema;
- a especificação da configuração do sistema (mapeamento de componentes de software no hardware e interconexão de componentes de software).

Este trabalho direciona conceitos presentes nas definições dessas linguagens de configuração para sua utilização dentro de uma estratégia de desenvolvimento que permite a determinação de componentes que possam ser projetados independentementes.

A estrutura de modelagem adotada é inspirada naquela preconizada em [McMenamin84, Ward85, Maffeo92], que tem sua base nos chamados **Métodos Estruturados** orientados a atividades (processos). Esses métodos têm privilegiado historicamente a identificação das funções componentes do sistema [DeMarco78, Gane79] mas, em suas versões mais modernas [Ward85, Yourdon88, Maffeo92], incorporam a modelagem paralela e simultânea da informação armazenada. Essa estrutura segmenta o resultado do processo de desenvolvimento em três modelos, com graus de abstração diferentes, que detalham toda a concepção e implementação do sistema. Existe uma relação de precedência lógica entre esses modelos que, entretanto, não define necessariamente a ordem cronológica mais adequada para sua construção. Detalhes referentes a esses modelos podem ser encontrados em [Maffeo91, Maffeo92 Sanchez94, Sanchez96]. São eles:

- **Modelo da Essência** (ME), representando o resultado da elicitação, análise e especificação dos requisitos essenciais (isto é, requisitos independentes de qualquer alternativa de implementação), constitui o modelo conceitual do sistema em questão [ISO695];
- **Modelo da Implementação** (MI), sucessor lógico do ME incorporando os requisitos essenciais ao design do sistema, caracterizado ao longo deste trabalho;
- **Modelo da Automação** (MA), sucessor lógico do MI incorporando o design da alternativa de implementação selecionada aos recursos tecnológicos de nível de abstração mais baixo (linguagem de programação, processadores,...).

Os produtos gerados nas diferentes etapas do processo de desenvolvimento estão indicados na Figura 1, onde cada modelo engloba o conteúdo do modelo que o precede logicamente e o símbolo "+" indica alternativa.

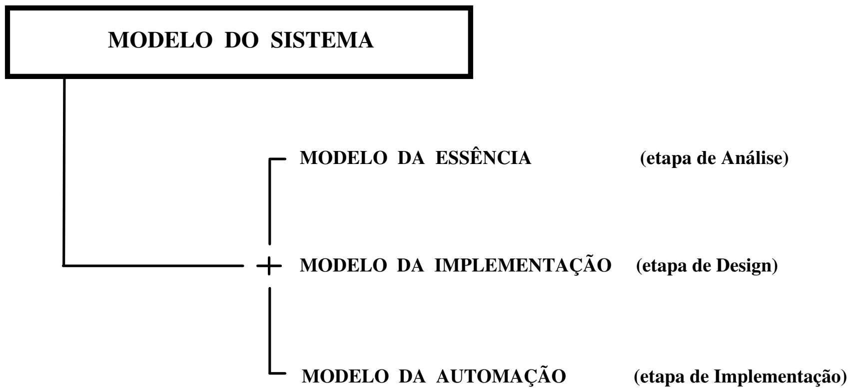

Figura 1 - Visão Estrutural do Processo de Modelagem

A seção 2 apresenta um resumo do processo de modelagem da essência. A Seção 3 introduz os conceitos básicos do método de design proposto. A seção 4 estabelece a ponte entre a modelagem da essência e a obtenção das classes que constituem a base para a definição dos subsistemas básicos que compõem o sistema. A seção 5 descreve a forma de validação do método de modelagem empregado, explicitando a análise das características de qualidade visadas. A seção 6 apresenta as conclusões do trabalho desenvolvido, focalizando principalmente a implantação de reúso segundo o processo de desenvolvimento proposto.

2. O Modelo da Essência

O método de modelagem aqui apresentado tem como primeiro ponto de segmentação/abstração o critério de "Essência versus Implementação" [Ward85, Maffeo92]. A modelagem da essência abstrai a forma de implementar o sistema — solução concreta de um problema computacional — e restringe-se ao registro dos seus requisitos de eficácia. Para representar adequadamente a solução abstrata, novamente utiliza-se o processo de segmentação/abstração, agora baseado em um critério de "pertinência ao sistema". Esse critério segmenta o Modelo da Essência em dois submodelos, o Modelo do Contexto e o Modelo do Comportamento, que representam, respectivamente, a definição das necessidades do ambiente externo — o enunciado do problema a ser resolvido pelo sistema computacional — e uma solução possível para esse enunciado, abstraída de referências a formas de implementação.

Cada um desses dois modelos é composto de submodelos complementares, construídos a partir de enfoques diferentes sobre o ambiente externo (modelagem do contexo) e sobre o sistema (modelagem do comportamento).

Assim, o **Modelo da Essência** [Maffeo92, Sanchez95b] apresenta a seguinte estrutura, representada esquematicamente na **Error! Reference source not found.** (a seta indica precedência lógica):

- **a. Modelo do Contexto**, composto de:
  - **Definição de Sistema**, descrevendo os objetivos e a operação "caixa-preta" do sistema. Emprega-se a Linguagem Natural com estrutura e vocabulário controlados;

- **Lista de Eventos Externos**, identificando os eventos que ocorrem no ambiente externo ao sistema de controle e suscitam reações planejadas por parte desse sistema. Essa lista constitui um instrumento básico de modelagem das necessidades a serem atendidas pelo sistema. Emprega-se a Linguagem Natural com estrutura e vocabulário controlados;
- **Esquema Transacional**, modelando a interface do sistema com as entidades externas com as quais interage diretamente.Emprega-se a ESML (Extended Systems Modeling Language) [Bruyn88, Maffeo91];
- **Esquema Semântico** [Maffeo90, Maffeo92], modelando individualmente cada transação implícita no Esquema Transacional e revelando a natureza e a composição dos fluxos e depósitos externos presentes nessa representação gráfica do contexto do sistema. Emprega-se um Diagrama de Entidades-Relacionamentos constituído por um conjunto de hierarquias semânticas do tipo "é-um" e "é-composto";
- **Esquema da Dinâmica** [Clemente92], modelando a dinâmica dos eventos externos de interesse para o sistema. Emprega-se a linguagem gráfica dos Diagramas de Estados e Transições [Ward85] e/ou das Redes de Petri Compactas [Heuser90, Heuser93].

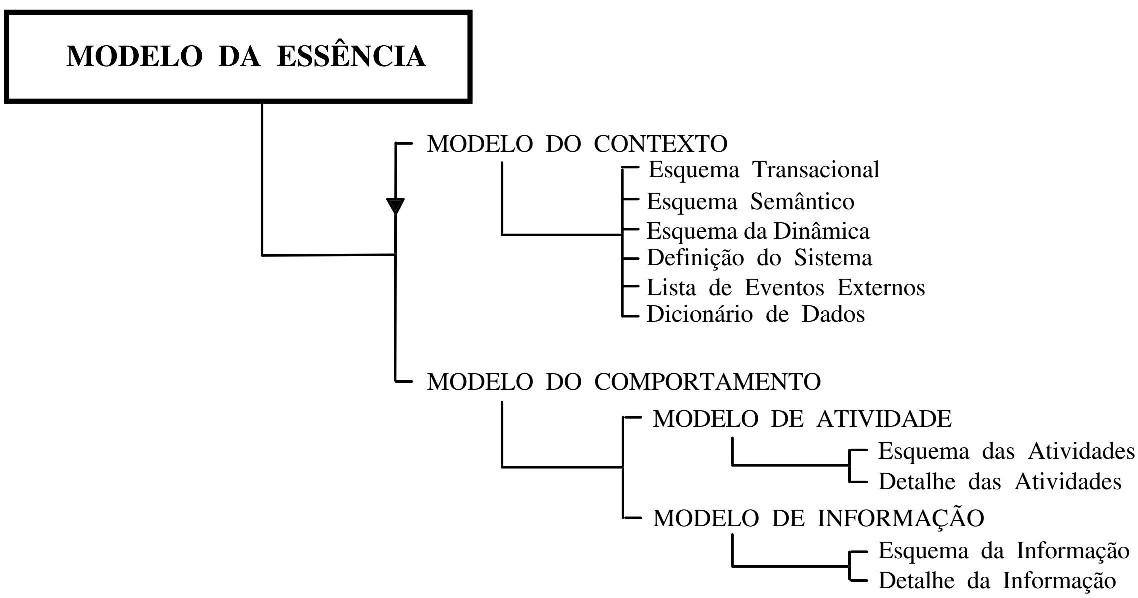

Figura 2 - Visão Estrutural da Modelagem da Essência

b.Modelo do Comportamento, composto de:

- **Esquema da Memória Essencial**, onde a linguagem de representação é o DER (Diagrama de Entidades e Relacionamentos) [Chen76], incluindo toda a parte esquemática da modelagem conceitual dos elementos funcionais passivos armazenados no sistema (tipos de entidade, tipos de relacionamento entre entidades e atributos);

- **Esquema das Atividades Essenciais**, que utiliza a ESML, modelando todas as atividades essenciais em um mesmo diagrama de rede, não hierárquico;
- **Representação Hierárquica das Atividades**, que utiliza a ESML, modelando os elementos funcionais ativos do sistema em termos de uma hierarquia de atividades até o nível das atividades operacionais primitivas, estas representando as funções do sistema que reagem à ocorrências de eventos externos e não são mais detalhadas graficamente;
- **Diagramas de Estados e Transições**, um para cada atividade de controle, constituindo sua especificação formal;
- **Listas de Pré/Pós Condições**, uma para cada atividade operacional primitiva, constituindo sua especificação formal.

Completando esses dois modelos, um Dicionário de Dados define a informação presente em sua parte esquemática e é ampliado e concluído durante a modelagem da implementação.

3. O Modelo da Implementação

O Modelo da Implementação constitui uma planta da automação agregando as características não-essenciais do sistema — aquelas relacionadas à forma de implementar o sistema — aos requisitos essenciais definidos no Modelo da Essência. Dessa forma são completados os requisitos necessários ao atendimento eficiente às necessidades do usuário. O Modelo da Implementação usado neste trabalho, proposto em [Sanchez95a, Sanchez96], é composto de três porções distintas:

- ♦ a definição dos **Requisitos de Concepção de Sistema**, que agrega requisitos não-essenciais relacionados primordialmente a desempenho e tolerância a falhas. Essa definição impõe restrições à tecnologia a ser empregada na implementação e é precondicão para a escolha das arquiteturas de hardware e de software básico;
- ♦ a definição da **Interface Usuário-Máquina** (IUM), que determina as características interativas do sistema e sua forma de operação. Implica selecionar e/ou construir os equipamentos de interface com os operadores e o software a ser incorporado ao sistema para estabelecer sua interatividade;
- ♦ o **design do sistema,** segmentado em termos de subsistemas que obedecem a um padrão de cooperação para atender aos requisitos essenciais e não-essenciais, aqui incluídos os requisitos para a comunicação com operadores.

A estratégia empregada para atingir o objetivo de reúso recomenda a divisão do processo de design do sistema em quatro etapas:

1. construção do **Projeto Básico**, que evidencia os **Subsistemas Básicos** (SBs)**,** suas **Portas** e **Conexões**, bem como a eventual necessidade de concorrência e sincronismo entre os serviços alocados a cada subsistema básico;

2. **especificação detalhada** de cada **subsistema básico**, que permite seu desenvolvimento autônomo e facilita sua recuperação para reúso;
3. **design** de cada subsistema básico como unidade independente de desenvolvimento, envolvendo implementação e reúso;
4. **configuração do sistema**, determinando-se a **configuração de software** (definição das portas e conexões associadas aos subsistemas básicos) e a **configuração de hardware** (mapeamento dos subsistemas básicos e seus processos no hardware).

A Figura 3 ilustra a estrutura do Modelo da Implementação.

O **Projeto Básico**, resultado de uma ação de síntese aplicada sobre o Modelo da Essência, contém a definição dos **subsistemas básicos** (SBs) que incorpora a especificação dos serviços prestados pelo sistema associados a cada um desses subsistemas. **Serviços** são especificados a partir de agregações de Eventos Externos definidos no Modelo do Contexto. **Subsistemas básicos,** determinados a partir de critérios de síntese aplicados ao Modelo do Comportamento e baseados no encapsulamento de informação (Tipos de Entidade) e de operações (Atividades Essenciais), englobam um conjunto resumido e bem definido de serviços passível de reúso em outros sistemas.

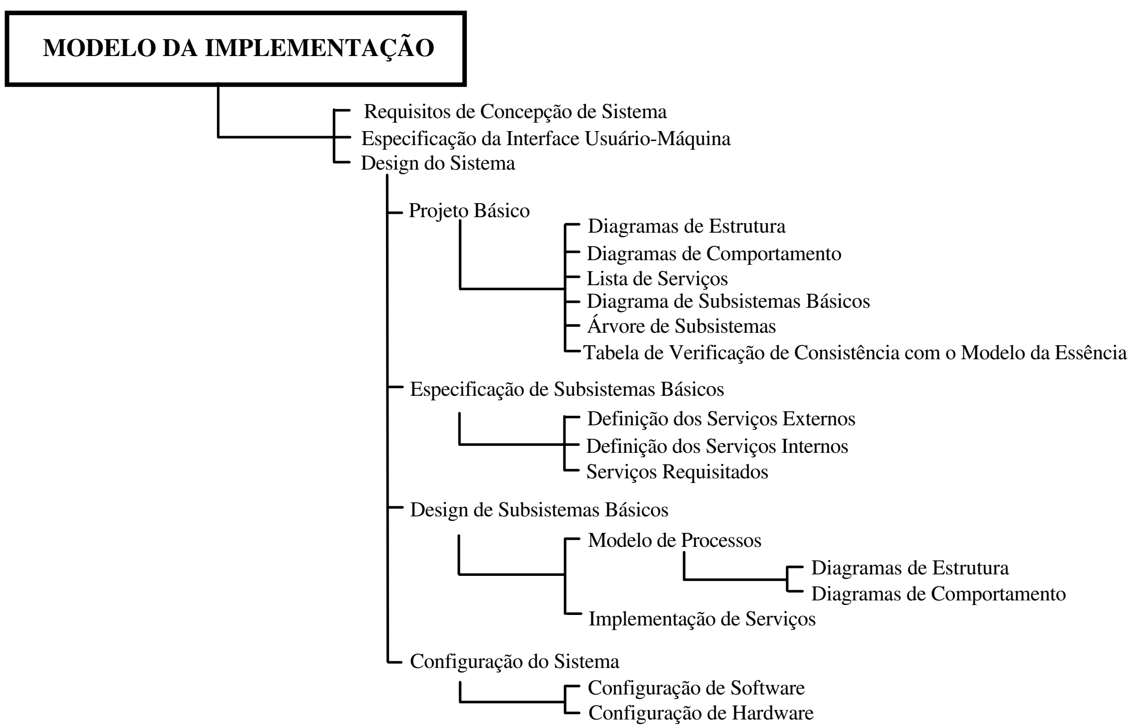

Figura 3 - Visão Estrutural da Modelagem da Implementação

Os **SB's** são **unidades de implementação** e **unidades de reúso**. Unidades de implementação que se prestam ao design, gerenciamento, produção e manutenção de grandes sistemas que encapsulam informação e operações de modo a garantir a inexistência de áreas de dados compartilhadas. Cada subsistema básico é uma unidade reusável de código fonte desenvolvida e mantida por uma equipe pequena, indivisível quando reusada.

Um sistema pode ser considerado como um prestador de serviços ao ambiente externo e esse enfoque determina um critério de segmentação — cada SB é responsável por um subconjunto dos serviços prestados pelo sistema —. Um **Serviço** é um grupo de funções (análogo a um conjunto de operações de uma classe da abordagem orientada a objetos) inter-relacionadas que compartilham propósitos comuns; em particular, gerenciam uma estrutura de dados originária da estrutura de informação encapsulada pela classe da qual deriva o SB. O conceito de **serviço** objetiva evidenciar o encapsulamento, visando ao reúso, de procedimentos de apoio às atividades do sistema que atendem diretamente às necessidades do ambiente externo. Convém ressaltar que uma necessidade do ambiente externo traduz-se pela ocorrência de evento externo e isso equivale a uma "requisição de serviço" feita por esse ambiente e direcionada a um componente específico do sistema. Um componente específico que interaja diretamente com o ambiente externo pode requisitar a execução de serviços a outros componentes que encapsulem procedimentos de apoio genéricos. Serviços diferentes requisitados pelo ambiente externo e executados por componentes específicos diferentes podem, durante sua execução, requisitar um mesmo serviço de apoio — caracterizando reúso, pelo próprio sistema, do procedimento de apoio encapsulado no componente requisitado —. Componentes que só encapsulam procedimentos de apoio reagem apenas a ocorrência de requisições de serviço efetuadas por componentes específicos responsáveis pela interação direta com o ambiente externo.

O sistema é representado como uma organização hierárquica de subsistemas com serviços alocados a cada subsistema. Cada folha da hierarquia é considerada **subsistema básico**. A segmentação do sistema em SBs obedece à restrição de que um serviço só pode aparecer em um dado subsistema. Caso algum outro subsistema necessite deste serviço deve pedi-lo ao subsistema ao qual o serviço esteja alocado. A dinâmica do sistema é caracterizada pela ativação desses serviços. Como os SB's encapsulam toda a informação tratada pelo sistema, nenhum dos níveis da organização hierárquica apresentará área de dados compartilhada.

O **Projeto Básico** especifica o conjunto de serviços pertencente a cada SB de uma forma ainda abstrata e sucinta. Entretanto, para que a implementação seja feita por equipes diferentes, cada SB necessita de uma especificação detalhada independente. Isto é, o serviço especificado no Projeto Básico ainda não estará descrito com o nível de detalhe necessário à implementação. A partir dele e do Modelo da Essência do sistema, será feita uma especificação detalhada do SB. Nessa especificação, para facilitar o reúso do subsistema básico, limites e restrições devem, sempre que possível, identificar o tipo de parametrização a ser empregado na implementação [Poulin94].

3.1 Cooperação entre Subsistemas

Visando proporcionar nível elevado de **reúso**, os SBs devem ser concebidos de modo que cada um tenha conhecimento apenas de si próprio — isto é, nenhum SB pode incorporar referências a outros SBs —. Para obter essa independência, cada SB deve possuir **portas,** de entrada e saída de dados e controle, que definam o protocolo de comunicação com o ambiente externo ao subsistema, regulando o **envio/recebimento de mensagens**. Essas mensagens são transportadas por uma **interface** composta de **canais de comunicação** que conectam **portas** de SBs. A comunicação e o sincronismo entre SBs se dá através dessa interface.

A definição precisa de um SB exige a especificação de:

- as entradas;
- as saídas;
- como, a partir das entradas, são produzidas as saídas. Incluindo eventuais restrições que o sistema deve satisfazer, essa definição contém a especificação abstrata de uma relação matemática, envolvendo essas entradas e saídas, que pode ser implementada alternativamente por mais de um procedimento.

No método de design aqui proposto, a definição das portas de um SB cumpre o requisito de especificar entradas e saídas. A cada porta associa-se um tipo de variável que a descreve, isto é, uma estrutura descrita a partir de dados atômicos. Para fins de comunicação, a definição, o design e a automação do SB deve fazer referência apenas a suas portas e nunca a outros SBs com os quais interaja. Através dessas portas são efetivamente ativados os serviços que o subsistema presta ao mundo exterior, enviados os dados necessários à execução de um serviço requisitado e recebidas as respostas correspondentes a sua execução. Dessa forma, é possível garantir elevado nível de **reúso** e, desde que seja respeitado o protocolo especificado para suas portas, subsistemas podem ser empregados em outros sistemas sem sofrer alterações, comunicando-se com subsistemas diferentes daqueles empregados na configuração do sistema para o qual foi desenvolvido.

3.2 Configuração

O design do sistema resulta em um conjunto de SBs independentes, cuja integração é efetuada através de uma **Linguagem de Configuração** com a função de definir as conexões entre portas de subsistemas, verificar a consistência entre as portas conectadas e mapear os processos de um dado SB na Arquitetura de Hardware. Ao empregar-se uma linguagem de configuração determina-se uma hierarquização vertical do software em duas camadas [Sanchez95a]. A camada de nível mais baixo (**Interface de Compatibilização**) depende somente das camadas inferiores (software básico e hardware) e oferece uma interface padrão de serviços prestados para a camada de nível mais alto (**aplicação**). A existência dessa Interface de Compatibilização permite que o software da aplicação seja projetado em maior nível de abstração, equivalente àquele empregado no design, o que é precondição para alcançar elevada portatilidade.

O conjunto de serviços oferecidos pela Interface de Compatibilização não depende da aplicação, mas sim das características do software básico (facilidades de comunicação/ sincronismo de processos), do hardware e dos conceitos, acima expostos, empregados no design. Assim sendo, a Interface de Compatibilização pode ser reusada para várias aplicações que façam uso das mesmas camadas inferiores. Seu emprego também garante ao software resultante grande capacidade de **reúso** pois:

- o uso de sincronismo e comunicação por mensagens, além de privilegiar a simplificação do processo de desenvolvimento do software de aplicação, permite o emprego maximizado dos conceitos de Encapsulamento de Informação ("information hiding");
- o emprego de um método de comunicação que estabelece, por um lado, conexão de portas definida através de linguagem de configuração e, por outro lado, envio/recebimento de mensagens onde cada subsistema referencia apenas suas próprias portas, permite o isolamento dos SBs e, dessa forma, garante-se sua independência. Assim, um mesmo serviço requisitado poder ser fornecido por vários SBs diferentes em função da configuração corrente do sistema;
- as características de comunicação entre SBs, impostas pela Interface de Compatibilização, possibilitam o uso de arquiteturas mistas de hardware e software básico bem como configurar, em um mesmo subsistema, diversos subsistemas já anteriormente implementados mesmo que com linguagens de programação diferentes.

4. O Processo de Síntese

Baseando-se no Esquema da Memória Essencial, no Esquema de Atividades Essenciais e na Organização Hierárquica de Atividades contidos no Modelo do Comportamento, é possível propor uma configuração de classes para o sistema e, conseqüentemente, também uma configuração de subsistemas básicos (SBs). Nesta seção, serão estabelecidas técnicas para a definição do design do sistema que devem mapear rigorosamente os requisitos (ações e informações) contidos no Modelo da Essência para o design do sistema, garantindo completeza e integridade.

4.1 Subsistemas Básicos x Objetos

Cada **subsistema básico** corresponde a uma alternativa de implementação para uma **classe de objetos** — uma estrutura de informação encapsulada com um conjunto de operações atuando sobre essa estrutura —.

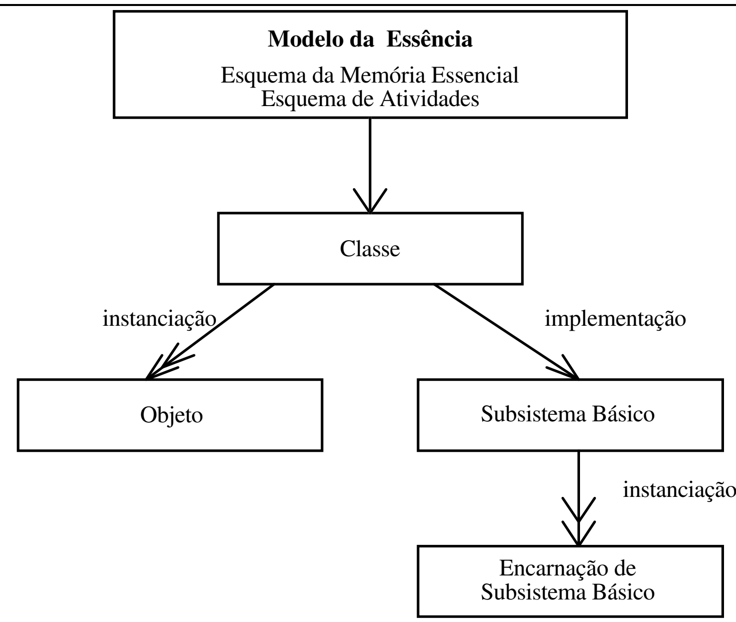

Figura 4 - Alternativas de síntese: objetos ou subsistemas básicos

O agrupamento de objetos em classes [Rumbaugh91] caracteriza um processo envolvendo abstração e síntese. **Abstração** de detalhes relacionados a entes do mundo real (objetos) dando origem a uma estrutura (classe) da qual cada ente é uma instância. **Síntese** pois uma classe encapsula uma estrutura de informação e operações que atuam sobre essa estrutura. Objetos são conceitualmente concorrentes.

O caminho que conduz, a partir do Modelo da Essência, à Configuração de Subsistemas Básicos passa pela determinação das classes que compõem o sistema. Devido a considerações de desempenho e a características do ambiente externo, o método apresentado neste trabalho adota, para classes, uma alternativa de implementação — subsistemas básicos — diferente da alternativa orientada a objetos (vide Figura 4). Essa escolha, entretanto, não prejudica o reúso desses SBs.

A implementação de uma classe através de um SB considera as características seqüenciais eventualmente inerentes à extração da informação associada a cada objeto. Assim, não são criadas instâncias distintas da estrutura de dados e dos processos associados a instâncias distintas de uma dada classe, criando-se apenas instâncias distintas da estrutura de dados que armazena os valores dos atributos da classe. O SB gerencia esse conjunto de instâncias de estruturas de dados com instâncias únicas de processos. Essa implementação implica o tratamento seqüencial de serviços associados à informação encapsulada. Isso é facilitado pelo conceito de porta pois, associada a cada processo existe uma única porta de entrada (ou entrada com resposta associada) onde são depositadas as requisições de serviço ao processo. O enfileiramento dessas requisições, para tratamento seqüencial, é realizado pela própria porta. Isso em nada altera a generalidade do conceito de classes. O número de instâncias gerenciadas por um mesmo SB pode ser determinado em tempo de geração do sistema ou, dinamicamente, em tempo de execução.

O uso de mais de uma encarnação do SB na configuração do sistema não é inibido. Essa multiplicidade é necessária caso exista concorrência na aquisição/tratamento da informação, o que sempre ocorre ao tratar-se mais de um sensor/atuador do mesmo tipo, e/ou para garantir o atendimento aos Requisitos de Concepção do Sistema dividindo-se a carga de processamento por mais de um processador.

A especificação de um sistema só deve estar comprometida com a utilização de processos concorrentes na medida que esse compromisso seja consistente com a simplicidade e a clareza do design. Geralmente, a existência de um grande número de processos concorrentes, além de tornar o design extremamente complexo, pode aumentar indevidamente a sobrecarga em tempo de execução decorrente de uma excessiva necessidade de comunicação e sincronização entre processos [Gomaa93]. A implementação de classes em termos de SBs, em vez de objetos (intrinsecamente concorrentes), visa abordar esse tipo de questão.

4.2 Identificando Subsistemas Básicos

A identificação das classes relevantes tem como ponto de partida o Esquema da Memória Essencial. Cada Tipo de Entidade (ou porção entidade de um Tipo de Relacionamentos com Atributos [Maffeo92]) corresponde a um depósito interno presente no Esquema de Atividades Essenciais. A primeira etapa para a obtenção das classes associa uma estrutura de informação encapsulada em classe a cada depósito.

Em seguida, são definidas as operações dedicadas a cada uma dessas estruturas, o que pode ser feito inicialmente a partir do Esquema de Atividades Essenciais e refinado posteriormente considerando-se as Listas de Pré- e Pós-condições. As operações encapsuladas em cada classe são extraídas do conjunto de Atividades Essenciais que fazem acesso à estrutura (depósito interno) por ela encapsulada. Todas essas atividades são candidatas ao encapsulamento pela classe.

Atividades Essenciais podem ser classificadas com relação ao tipo de acesso que fazem a um dado depósito interno:

- Atividade Primária: faz acesso de escrita ao depósito para inserção ou remoção de entidades;
- Atividade Secundária: faz acesso de escrita ao depósito apenas para alteração do estado de entidades;
- Atividade Fundamental: faz apenas acesso de leitura ao depósito.

Normalmente uma Atividade Essencial faz acesso a mais de um depósito interno e a decisão relativa a qual classe deve encapsular essa atividade baseia-se em:

- uma atividade primária relativa a uma dada estrutura de informação deve ser instalada na classe que encapsula essa estrutura;
- uma atividade secundária relativa a uma dada estrutura de informação, que não seja atividade primária relativa a qualquer outra estrutura, deve ser instalada na classe que encapsula aquela estrutura;
- uma atividade fundamental relativa a uma dada estrutura de informação, que não seja atividade primária ou secundária relativa a qualquer outra estrutura, deve ser instalada na classe que encapsula aquela estrutura.

Essa alocação inicial deve ser revista pois, em geral, apresenta uma mesma atividade essencial instalada em mais de uma classe. Essa revisão tem como critério básico a coesão funcional de cada classe. Assim sendo:

- é removida da classe onde foi inicialmente instalada toda atividade primária cujo objetivo não esteja relacionado ao objetivo de armazenamento da estrutura de informação que a classe encapsula. Caso a atividade removida não seja primária em relação a qualquer outra estrutura de informação, deve ser instalada na classe que encapsula uma estrutura em relação à qual seja secundária, ou na ausência desta, fundamental;

- devem ser buscadas agregações que permitam a uma classe encapsular estruturas de informação associadas a tipos de entidades associados por tipos de relacionamentos no Esquema da Memória Essencial juntamente com (parte de) o conjunto de atividades essenciais que fazem acesso aos depósitos internos correspondentes.

Se subsistirem atividades essenciais instaladas em mais de uma classe, deve ser identificado o objetivo principal da atividade e a alternativa adotada é instalá-la apenas na classe que encapsula a estrutura de informação mais compatível com esse objetivo. Uma atividades de controle, que (por definição) não faz acesso a depósitos (de dados), deve ser instalada na classe que encapsule o maior subconjunto de atividades operacionais por ela controladas. Ambos os critérios buscam maximizar a coesão funcional.

Existe ainda necessidade de refinamento visando respeitar critérios de coesão funcional relacionando operações que atuam sobre estruturas de dados encapsuladas em classes distintas. Isso conduz à alocação de atividades e agregação de informações de forma a:

- alocar a uma mesma classe atividades cujos conjuntos de operações obedecem à dinâmica definida por uma única função de controle;
- agregar classes correspondentes a Tipos de Entidade associados através de relacionamento com sentido único (acesso indireto ao Tipo de Entidade fraco)
- agregar classes correspondentes a Tipos de Entidade associados através de um Tipo de Relacionamento com Atributos, entre si e com a classe correspondente à Porção Entidade desse tipo de relacionamento, quando as operações que atuarem sobre essas estruturas forem fortemente coesas;
- agregar classes relacionadas ao tratamento de um único dispositivo de hardware (sensor/atuador monitorado/controlado pelo sistema).

Alocações e agregações podem ser realizadas visando aumentar o grau de reúso das classes resultantes. Assim sendo, elementos funcionais (passivos e ativos) que, quando instanciados, sejam sempre empregados em conjunto devem constituir uma única classe. O critério de reúso de classes determinará nova revisão de forma a:

- desagregar classes para explicitar características não reusáveis de operações (devido ao fato de seus requisitos não atenderem às expectativas de outros clientes/usuários), isolando-as de operações reusáveis;
- desagregar classes para explicitar características reusáveis mas que não necessitem ser sempre usados em conjunto;
- eliminar possíveis redundâncias entre classes. Por exemplo, um conjunto de serviços que sejam comuns a várias classes deve ser encapsulado isoladamente em uma classe específica (classe utilitária).

Critérios específicos de reúso não estão diretamente relacionados à dedução das classes e foram adicionados apenas para complementar a definição do método de design. No entanto, os critérios sobre os quais baseia-se a dedução das classes, direcionados primordialmente para a obtenção de elevada modularidade, manutenibilidade e portatilidade, implicam, devido à não-ortogonalidade desses atributos de qualidade em relação ao grau de reúso, o aumento da capacidade de reúso das classes obtidas.

Em particular, a implementação de classes em termos de SBs deve dar origem, preferencialmente, a subsistemas totalmente reusáveis ou não-reusáveis. A aplicação de critérios específicos relacionados a reúso possibilita um refinamento das classes obtidas a partir da aplicação das diretivas, expostas precedentemente, para a derivação de um design orientado a SBs fundamentado na modelagem da essência. Esse refinamento deverá implicar um aumento do grau de reúso das classes refinadas.

A identificação de características reusáveis em conjunto e das não-reusáveis pode requerer um processo formal de **Análise de Domínios**, que envolve a descoberta de pontos comuns e variantes de sistemas em um domínio [Prieto-Diaz91]. Essa análise constitui uma tentativa de identificar objetos, operações e relações que os especialistas do domínio percebem ser importantes [Arango89]. Difere dos outros métodos basicamente por visar à modelagem de famílias inteiras de sistemas.

Para facilitar também essa revisão final das classes derivadas da essência com o objetivo de aumentar seu potencial de reúso em outros sistemas, devem ser empregadas heurísticas [Sanchez95a], algumas originárias da literatura de orientação a objetos e outras baseadas em conceitos associados à programação estruturada. Como exemplo, pode-se citar a utilização de **otimização controlada** [Weide94], onde a heurística de separar algoritmos em classes independentes foi demonstradamente eficiente para apoiar o reúso “caixa-preta” de componentes de software, aumentando a flexibilidade funcional (componente adaptável para prestar o mesmo serviço a classes diferentes) do sistema e o desempenho do componente (substituição de alternativas de implementação, melhorando a forma de implementar o serviço). Essa heurística vem reforçar os critérios de desagregar classes que não serão reusadas em conjunto e de eliminar possíveis redundâncias. Outro exemplo é a heurística de separar em uma classe independente uma atividade de controle que coordene um conjunto de interações entre diversas classes em função de seus estados internos [Sanchez95a, Bulman89]; trata-se aqui de uma generalização da heurística de programação estruturada que determina separar operação e controle [Page-Jones88]. Ambas as heurísticas consideradas reforçam o critério de isolar características não-reusáveis de características reusáveis.

Frakes [Frakes94] afirma que, apesar de alguns princípios gerais terem sido definidos em [Tracz90], ainda não existe um método de design para componentes reusáveis. Princípios importantes de Engenharia de Software — tais como abstração/encapsulamento de dados e otimização controlada — são críticos para permitir o design de componentes reusáveis. Deve ser ressaltado que os princípos básicos estabelecidos por Frakes para alcançar bons componentes reusáveis são também utilizados no método de design proposto neste trabalho.

Entretanto, convém ressaltar que, se restrito aos aspectos técnicos, nenhum método de desenvolvimento de sistemas pode garantir por si só a obediência aos padrões que o caracterizam. Essa obediência só pode ser imposta através de um rígido controle gerencial, exercido durante todo o processo de desenvolvimento, através do qual seria garantida a geração de produtos com qualidade assegurada por construção. No entanto, nenhum processo gerencial deve ser implantado sem que haja um suporte metodológico para a vertente técnica do processo de desenvolvimento.

A técnica de design proposta neste trabalho visa fornecer esse suporte para que a otimização dos atributos de qualidade prefixados possa ser alcançada.

Quando não existir concorrência na requisição de serviços a instâncias de uma dada classe, essa classe deverá ser implementada como um SB que gerencia um conjunto de instâncias da estrutura de informação nela encapsulada. Havendo concorrência na requisição de serviços, o SB que constitui a implementação da classe considerada deverá ser distintamente instanciado dando origem a diferentes encarnações (ver Figura 4). Cada encarnação tratará concorrentemente uma requisição de serviço evidenciando-se, dessa forma, o paralelismo existente entre as requisições consideradas.

Em seguida, deve ser definida a forma de ativação de cada uma das operações de um dado SB através da interface que o conecta aos demais SB's e ao ambiente externo ao sistema. As portas de um SB devem atender às necessidades de comunicação entre atividades encapsuladas em SBs distintos e às necessidades de comunicação entre cada SB e o ambiente externo.

Portas de comunicação com o ambiente externo resultam das conexões existentes entre as atividades essenciais encapsuladas pela classe e esse ambiente e da concorrência existente entre operações. Uma conexão entre duas atividades essenciais não encapsuladas em um mesma classe dá origem a uma requisição de serviço entre as classes às quais pertencem essas atividades e à existência de uma porta de saída no SB cliente e de uma porta de entrada no SB servidor. O número de portas e os tipos de mensagem alocados a cada porta dependem das características de concorrência existente entre as atividades essenciais encapsuladas pela classe que dá origem ao SB. Essas características estão definidas em Diagramas de Estado e Transição que especificam as atividades de controle do Esquema de Atividades Essenciais. As portas são agregadas em função da não-concorrência existente entre os serviços ativados por uma dada porta de entrada, de forma a:

- reunir requisições de serviço com características não-concorrentes em uma mesma porta, que ficará associada a um conjunto de serviços. Os tipos de mensagem diferentes equivalentes a requisições de serviços diferentes ativados através dessa porta são compatibilizados em um único tipo de mensagem associado à porta;
- associar serviços concorrentes a portas de entrada distintas (isto é, o número de portas de entrada deve igualar o número de conjuntos concorrentes de serviços não-concorrentes).

Devem constituir serviços não-concorrentes os tratamentos diferenciados relativos a valores específicos de campos de dados da mensagem associada à porta. Além disso, dados internos às mensagens não devem determinar a lógica interna aos serviços. Trata-se de critério suplementar que melhora a visualização dos serviços oferecidos por uma classe e, conseqüentemente, sua capacidade de entendimento para futuro reúso. Esse critério equivale a incorporar a heurística da programação estruturada de não empregar acoplamento de controle [Page-Jones88].

As classes obtidas a partir do Modelo do Comportamento são implementadas através de SBs que incorporam as atividades essenciais do sistema. A esses subsistemas devem ser acrescentadas operações sobre as estruturas de informação características da alternativa de implementação adotada. Ao conjunto de subsistemas devem ser acrescentados subsistemas contendo apenas atividades não-essenciais, tais como como as associadas à implementação da Interface Usuário Máquina — IUM — (entrada, interpretação de comando e apresentação de resultados). O SB que implementa a IUM pode vir a ter uma implementação complexa que fica totalmente tratada no design desse subsistema básico.

Trabalhos na área de Abstract Data Views (ADV) [Cowan95] comprovam a possibilidade e evidenciam as vantagens de especificar independentemente o design da IUM em relação ao design da aplicação. O método de design aqui proposto, empregando ferramentas e técnicas de modelagem diferentes, também prescreve a separação entre IUM e aplicação. O método deve dar origem a um conjunto de subsistemas que trata apenas da aplicação e é independente do subsistema que trata da IUM. Por compatibilidade, outros SBs são também incorporados para encapsular o uso de utilitários especiais como, por exemplo, um gerenciador específico de banco de dados. Na realidade, dependendo das facilidades oferecidas pelo utilitário empregado, pode ser introduzido apenas um "código cola" ("glue code"), usado para garantir a compatibilidade de componentes [Neighbors94], adaptando seu uso ao método de design aqui proposto. Esse "código cola" seria uma pequena porção de código que define portas e serviços associados, cuja implementação é a chamada correta das funções do utilitário. Isso é justificável pois o resultado obtido:

- encapsula alternativas de implementação;
- garante uma arquitetura do tipo cliente-servidor;
- permite que esse utilitário, encapsulado em um SB, execute em processador diferente daqueles onde executam os subsistemas que utilizam seus serviços. Assim, esse procedimento isola características da arquitetura de hardware, bem como características da configuração de subsistemas utilizada nessa arquitetura, em relação aos aspectos relativos a uma opção específica de implementação de software.

O acréscimo desses novos subsistemas, relativos a alternativas específicas de implementação, determina outra revisão das definições dos SBs já gerados e das definições de suas portas, visando identificar a necessidade de novas portas devido à incorporacão dos serviços característicos da implementação.

O sistema desenvolvido compõe-se de SBs e o reúso destes em outro sistema a ser futuramente desenvolvido pode gerar a necessidade de acrescentar novas portas ou novos serviços através de portas já existentes. É importante notar que:

- em uma dada configuração de um sistema, é possível que nem todos os serviços de um SB sejam solicitados. Nesse caso, as portas associadas a um serviço não solicitado ficam inoperantes (não são conectadas a portas de saída de outros subsistemas);
- o acréscimo de novas portas de entrada (visando ao atendimento a novas requisições concorrentes), ou de novos serviços não-concorrentes a portas já existentes de um dado SB, aumenta seu escopo de atuação e caracteriza a potencialidade de reúso do SB.

Certamente, quase todos os SBs requisitam serviços a outros subsistemas. Reusar um subsistema implica existir na configuração do sistema algum subsistema que preste o serviço requisitado. Em princípio, isso deve refletir o domínio do problema e ocorreria em qualquer método de design orientado a objetos. Por outro lado, aqui é garantida a independência entre subsistemas, imposta pelo conceito de portas e de configuração independente. Por exemplo, reusar um subsistema <u>A</u> que presta um serviço <u>a</u> cuja execução requisita um serviço <u>b</u> implica:

- a existência, na configuração do novo sistema, de **qualquer** subsistema que preste o serviço b requisitado durante a execução do serviço a, não sendo necessário que esse subsistema seja o mesmo usado na configuração de origem do subsistema A;
- a não conexão da porta de saída do subsistema A que requisita o serviço b, caso o serviço a fique desativado durante a execução do novo sistema onde o subsistema A será reusado (isso seria análogo ao reúso de um chip em uma placa de hardware sem que todos os seus pinos estejam conectados).

5. Validação do Método

O método de desenvolvimento aqui descrito foi avaliado e refinado através de um experimento controlado envolvendo o desenvolvimento do subsistema de monitoramento e controle (Sistema para **Co**ntrole e **Mon**itoramento de **Li**tografia por **F**eixe de **E**létrons - **COMONLIFE**) para um sistema processo-controle realista com características de tempo-real. O sistema escolhido para o experimento, COMONLIFE, possui complexidade intrínseca e porte suficientes para testar o método e é compatível, em termos de prazo e custo, com um processo de denvolvimento a ser efetuado em ambiente de pesquisa acadêmica. Produtos desse experimento, reportando integralmente a modelagem da essência e a da implementação, encontram-se em [Sanchez95b] e [Sanchez95c]. Até o presente momento, a implementação efetiva (Modelo da Automação) limitou-se a um simulador para o COMONLIFE. Verificou-se que a técnica de design aqui proposta, implicando a derivação do design a partir do Modelo da Essência do COMONLIFE, não difere substancialmente da que seria empregada para gerar uma implementação orientada a objetos. No entanto, distintamente das abordagens correntes da orientação a objetos, a utilização

- do conceito de portas, inserido em um contexto de troca de mensagens e
- de um procedimento orientado para configuração independente de subsistemas (ao invés da interação entre subsistemas através de invocações de serviços inseridas diretamente no código)

permitiu maior flexibilidade ao processo de desenvolvimento e implica maior capacidade de reúso sem qualquer necessidade de introduzir alterações nos SBs obtidos.

Um método de design, direcionado a Sistemas de Informação e, portanto, em princípio adaptável para tratar a mesma classe de sistemas aqui considerada, baseado na identificação de classes a partir de um processo de síntese aplicado ao resultado da Análise Estruturada clássica, pode ser encontrado em [Bulman89]. As diferenças básicas em relação ao método proposto neste trabalho são:

- ao usar Análise Estruturada clássica, onde não é empregada uma Lista de Eventos Externos, não explora a modelagem conceitual da forma aqui preconizada e, por essa razão, restringiria perigosamente o número de perspectivas de análise sobre o processo a ser controlado;
- não apresenta critérios detalhados para a obtenção das classes, fundamentando-a basicamente na relação estímulo-resposta — a qual, quando possui uma estrutura complexa pode sugerir candidatos a objetos — representada no análogo ao Esquema Transacional do Modelo do Contexto .

Os resultados correntes do experimento que vem sendo conduzido já permitem afirmar que o emprego do método de design aqui apresentado proporciona meios para atingir objetivos de qualidade — modularidade, manutenibilidade, portatilidade e reúso — que constituem a principal motivação do trabalho de pesquisa empreendido.

No que se refere à modularidade e à manutenibilidade, essa adequação pôde ser fundamentada experimentalmente no momento em que, devido à indisponibilidade de equipamento relacionado ao processo a ser controlado, decidiu-se implementar um simulador para o COMONLIFE. O Modelo da Essência e o Projeto Básico haviam sido elaborados visando a um sistema real. Não obstante, para especificar a implementação do simulador, nada foi alterado nesses modelos quando subsistemas específicos destinados ao tratamento de sensores/atuadores reais foram substituídos por simuladores desses sensores e atuadores. Isto é, o Modelo da Essência completo e o Projeto Básico fundamentam implementações necessariamente distintas: a do simulador e a do sistema real. Apenas as especificações e o design dos SBs relacionados a sensores e atuadores foram alterados, mantendo-se a interface original. Isso garantiu um impacto de alteração limitado nos modelos desenvolvidos quando decidiu-se pela mudança de funcionalidade associada ao sistema — de 9 SBs, 5 tiveram suas especificações alteradas, e essas alterações limitaram-se à forma de interfaceamento com o ambiente externo, não tendo ocorrido qualquer alteração nos modelos de mais alto nível de abstração (Modelo da Essência e Projeto Básico) —.

Quanto à portatilidade, a implementação do simulador permite afirmar a invariância do código gerado para os SBs em relação à arquitetura de hardware e de software básico. Utilização específica de referência ao ambiente de desenvolvimento só ocorreu na implementação da IUM, que emprega chamadas a um utilitário desse ambiente e, portanto, apenas esse subsistema teria de ser alterado ao migrar-se o software desenvolvido para outra plataforma de desenvolvimento.

Quanto ao reúso, pode ser seguramente inferida, sobretudo em função da não-ortogonalidade das quatro métricas de qualidade visadas pelo método, a viabilidade do reaproveitamento de SBs no desenvolvimento de sistemas que compartilhem o domínio do COMONLIFE.

6. Conclusão

Considerando-se primordialmente a **implantação do reúso** segundo o método aqui apresentado pode-se concluir que, após o desenvolvimento de alguns sistemas a área de Informática da organização poderá dispor de uma biblioteca significativa de SBs reusáveis. Convém, entretanto, ressaltar que a construção e o gerenciamento de uma biblioteca de componentes reusáveis apresentam um conjunto de desafios, bem explicitados em [Neighbors94, Biggerstaff94].

Durante a concepção do Projeto Básico para um sistema novo, deve ser feita uma pesquisa na biblioteca de SBs para determinar os já existentes cujas características, obtidas a partir de suas especificações, sugiram reúso. A decomposição do sistema em subsistemas deve ser feita com esse conhecimento prévio e direcionada ao aproveitamento máximo dos SBs existentes, na forma de reúso “caixa-preta” [Neighbors94] pois nesse caso os SBs podem ser interconectados sem modificação. Um sistema pode até vir a ser construído totalmente pela interconexão de SBs já desenvolvidos, tal como hoje em dia constrói-se um módulo de hardware pela interligação de componentes integrados disponíveis no mercado. Pequenas alterações nos subsistemas, principalmente no que diz respeito a acréscimo de portas ou alterações nos serviços alocados a uma dada porta, conforme mencionandas na seção 4, permitem também um reúso "caixa-branca" [Neighbors94] de SBs pois sua documentação incorpora à especificação as decisões de design.

Implementar um programa de reúso sistemático com sucesso é tarefa difícil e de alto risco. Entretanto, não fazê-lo pode também representar alto risco. Inicialmente, a implantação de um método com reúso sistemático implica aumentar custos, não recuperáveis em caso de insucesso. Mas, se a concorrência o fizer com sucesso isso implicará perda de mercado pois os ganhos de produtividade e qualidade atuarão em desfavor da organização. Nessa área também não existe receita de sucesso, cada organização irá lidar com problemas técnicos e não-técnicos (culturais) e deverá definir sua próprias necessidades, métricas de reúso, benefícios esperados, identificar e remover impedimentos e gerenciar o risco [Frakes94].

Para a implantação da política de reúso, é necessário que o método de desenvolvimento esteja integrado a uma política administrativa que:

- incentive o reúso de SBs, pois programadores são inclinados a produzir suas próprias soluções ao invés de usar as já existente [Panel94];
- garanta a implementação de SBs reusáveis, analisando-os quanto a métricas de reúso como as descritas em [Poulin94] (por exemplo: ser o mais geral possível, não empregar variáveis globais, ser parametrizável, possuir boa documentação);
- controle o conjunto dos SB já existentes e efetive sua aplicação nos novos sistemas. A identificação de subsistemas que mais se adaptem a um ou outro requisito não é tarefa fácil. Suporte automatizado, como um sistema de biblioteca com assistente, que apóie a busca de um subsistema com uma dada funcionalidade [Merkl94], pode ajudar bastante;
- gerencie as diversas versões de SBs e os sistemas nos quais estão empregadas. Qualquer alteração em um desses subsistemas implica a necessidade de sinalizar o fato às equipes responsáveis pela manutenção dos outros sistemas que o utilizam.

No entanto, poucas organizações estão preparadas para suportar o ônus da implementação e validação de componentes reusáveis. Adicionar generalidade aos subsistemas e documentá-los dentro de um formato padrão para pesquisa futura é raramente feito devido a pressões de prazo [Panel94]. A implantação do reúso exige, portanto, o comprometimento continuado dos níveis mais altos de gerência da organização pois requer anos de investimento antes de existir retorno compensador e envolve mudanças gerenciais estruturais sem as quais nenhuma atividade relacionada a reúso sistemático poderá ter sucesso [Frakes94].

Considerando que cada SB é primordialmente derivado a partir de um Tipo de Entidade e de um conjunto de Atividades Essenciais, o reúso de um SB pode implicar o reúso de toda a modelagem baseada nos eventos externos tratados pelas atividades essenciais encapsuladas no SB. No Modelo do Contexto, submodelos como o Esquema Semântico, a Descrição de Operações e o Esquema da Dinâmica são totalmente rastreáveis a partir de eventos externos. No Modelo do Comportamento, todos os submodelos são também rastreáveis a partir de eventos externos.

Nessas condições, observa-se uma contribuição para a reengenharia de sistemas de software. Num processo de reengenharia visando à recuperação da essência, ao ser identificado um subconjunto de eventos externos e o SB que reage a sua ocorrência, é possível a identificação de outros eventos externos, também tratados pelo mesmo SB, relevantes para o Modelo da Essência mas ainda não identificados. É claro que o reúso pode ser pensado em nível de abstração ainda mais elevado e aplicar-se à própria modelagem do contexto contida no Modelo da Essência. Em particular, para problemas análogos, subconjuntos da Lista de Eventos Externos — e todos os demais componentes do Modelo do Contexto associados a esses subconjuntos, bem como toda a cadeia de reações subseqüentes especificada no Modelo do Comportamento — podem ser reusados na modelagem conceitual do novo sistema.

A definição de um método de busca para reúso de toda a modelagem baseada em eventos externos exigirá uma técnica que relacione de forma eficiente:

- subconjuntos de eventos externos;
- os componentes do Modelo da Essência associados a esses subconjuntos ou deles derivadas, incluindo requisitos essenciais funcionais (ativos e passivos) representados no Modelo do Comportamento;
- SBs que os implementam.

Bibliografia

Arango89	Arango G.; Domain Analysis: From Art Form to Engineering Discipline; SIGSOFT Engineering Notes, vol. 14(3); maio de 1989, p. 153.
Biggerstaff94	Biggerstaff T.J.; The Library Scaling Problem and the Limits of Concrete Component Reuse; Proceedings of Third International Conference on Software Reuse - Advances in Software Reusability, Rio de Janeiro; novembro de 1994.
Booch94	Booch G.; Object-Oriented Analysis and Design with Apllications; The Benjamin Cummings, Second Edition; 1994.
Bruyn88	Bruyn W., Jensen R., Keskar D. e Ward P.T.; "ESML: An Extended System Modelling Language Based on Data Flow Diagram"; ACM Sigsoft, Software Engineering Notes, 13(1): 58-62; janeiro de 1988.
Bulman89	Bulman D.M.; An Object-Based Development Model; Computer Language: 49-59; agosto de 1989,.
Bustard88	Bustard D., Eldore J., Welsh J.; Concurrent Program Structure; Prentice Hall; 1988.
Chen76	Chen P.P.S.; The entity-relationship model - toward a unified view of data; ACM Transactions on Database Systems, 1(1): 9-36; 1976.
Clemente92	Clemente K.; Modelagem de Sistemas Sócio-Técnicos, Estudo de Caso de um Piloto Automático para Automóvel; Tese de Mestrado, Departamento de Engenharia Elétrica, PUC-Rio; abril de 1992.
Cowan95	Cowan D.D., Lucena C.J.P.;Abstract Data Views: An Interafce Specification Concept to Enhance Design for Reuse; IEEE Transactions on Software Engineering, vol. 21, nº 3; 1995.

DeMarco78	DeMarco T.; Structured Analysis and System Specification; Englewood Cliffs, N.J., Prentice Hall; 1978.
DeRemer76	DeRemer F., Kron H.; Programming-in-the-Large versus Programming-in-the-Small, IEEE Transactions on Software Engineering, junho de 1976; pp 321 - 327.
Frakes94	Frakes W.B.; Success Factors of Systematic Reuse; IEEE Software; setembro de 1994.
Gane79	Gane, C., Sarson T.,; Structured Systems Analysis: Tools and Techniques; Fourth Edition; Englewood Cliffs, N.J., Prentice Hall; 1979.
Gomaa93	Gomaa H.; Software Design Methods for Concurrent and Real- Time Systems; Addison Wesley; 1993.
Heuser90	Heuser C.A.; Modelagem Conceitual de Sistemas - Redes de Petri; Publicação da V EBAI; 1990
Heuser92	Heuser C.A. e Richter G.; Constructs for Modeling Information Systems with Petri Nets; Application and Theory of Petri Nets, 13th International Conference Sheffield, UK, 22- 26, Proceedings, junho de 1992.
ISO695	Griethuysen J.J.(editor); Concepts and Terminology for the Conceptual Schema and the Information Base; International Standartization Organization (ISO), ANSI, Secretariat ISO/TC97/SC5; New York, nº 695.
Jackson83	Jackson M; System Development; Englewood Cliffs, N.J.: Prentice Hall; 1983.
Lauer79	Lauer H.C., Satterthwaite; The Impact of MESA on System Design; Proceedings of th 4th International Conference on Software Engineering; Munich, Germany; setembro de 1979; pp.174 -182.
Maffeo90	Maffeo B., Ritto A.C.A.; O Esquema Semântico no Modelo do Contexto de um Sistema Computacional; XXIII Congresso de Informáica da SUCESU; 1990.
Maffeo91	Maffeo B.; ESML: Uma Revisão de Apresentação, Estrutura, Notação e Conteúdo; Monografias em Ciência da Computação 1/91, Departamento de Informática, PUC-Rio;1991.
Maffeo92	Maffeo B.; Engenharia de Software e Especificação de Sistemas; Editora CAMPUS; 1992.
Mcmenamin84	McMenamin M., Palmer J. F.; Essential Systems Analysis; Yourdon Press; 1984.
Merkl94	Merkl W.; Learning de Semantic Similarity of Reusable Software Componens; Proceedings of Third International Conference on Software Reuse - Advances in Software Reusability, Rio de Janeiro; novembro de 1994.
Neighbors94	Neighbors J.M.;An Assessment of Reuse Technology After Ten Years; Proceedings of Third International Conference on Software Reuse - Advances in Software Reusability, Rio de Janeiro; novembro de 1994.
Page-Jones88	Page-Jones M.; The Practical Guide to Structured Systems Design,

Second Edition; Englewood Cliffs, N.J.; Prentice-Hall; 1988.

Panel94 Griss M.L., KozaczynskiW. V., Wasserman A. I., Jette C., Troy R.; Panel: Object-Oriented Reuse; Proceedings of Third International Conference on Software Reuse - Advances in Software Reusability, Rio de Janeiro; novembro de 1994.

Parnas72 Parnas, D.; On the Criteria for Decomposing a System into Modules; Communications ACM; dezembro de 1972.

Poulin94 Poulin J.S.; Measuring Software Reusability; Proceedings of Third International Conference on Software Reuse - Advances in Software Reusability; novembro de 1994.

Prieto-Diaz86 Prieto-Diaz R., Neighbors J.; Module Interconnection Languages; Journal of System Sciences; vol. 6, nº 4; novembro de 1986; pp 307 - 334.

Prietro-Diaz91 Prietro-Diaz R., Arango G.; Domain Analysis and Software Systems Modeling; IEEE Computer Society Press;1991.

Rumbaugh91 Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W.; Object-Oriented Modeling and Design; Prentice-Hall; 1991.

Sanchez94 Sanchez M. L., Maffeo B., Leite J. C. S. P. ; Ferramentas e Técnicas para a Modelagem da Essência de Sistemas de Tempo-Real para Controle e Monitoramento de Processos; Anais do 10º Congresso Brasileiro de Automática e do 6º Congresso Latino Americano de Controle Automático, Rio de Janeiro; setembro de 1994.

Sanchez95a Sanchez M. L., Maffeo B; Sistemas de Tempo-Real para Controle de Processos - Design Orientado a Encapsulamento de Dados e a Troca de Mensagens entre Subsistemas Autônomos; Série Monografias em Ciência da Computação 13/95, Departamento de Informática, PUC-Rio; 1995.

Sanchez95b Sanchez M. L., Maffeo B; Modelagem da Essência de um Sistema de Litografia por Feixe de Elétrons - COMONLIFE; Série Monografias em Ciência da Computação 29/95, Departamento de Informática, PUC-Rio; 1995.

Sanchez95c Sanchez M. L., Maffeo B; Design Baseado em Troca de Mensagens entre Subsistemas Autônomos de um Sistema para Controle e Monitoramento de um Processo de Litografia por Feixe de Elétrons - COMONLIFE; Série Monografias em Ciência da Computação 42/95, Departamento de Informática, PUC-Rio; 1995.

Sanchez96 Sanchez M.L.; Um Método de Design de Sistemas de Tempo-Real Fundamentado por Experimentação; Tese de Doutorado; PUC-Rio; 1996.

Simpson86 Simpson H.; The Mascot Method; IEE/BCS Software Engineering Journal, vol. 1, nº 3; maio de 1986, pp 103-120.

Sloman85 Sloman M., Kramer J., Magge J.; The CONIC Toolkit for Building Distributed Systems; 6ª IFAC Distributed Computer Control Systems Workshop; Monterey, California, Pergamon Press; maio de 1985.

Tracz90 Tracz W.; The 3 Cons of Software Reuse; Proceedings of Third annual Reuse Workshop; CASE Center tech report, Syracuse University, Syracuse, N.Y.; 1990.

Veldwiljk94	Veldwijk R.J. e Boogaard M.; Assessing the Software Crisis: Why Information Systems are Beyond Control; Information Sciences 81; 1994.
Ward85	Ward P.T. e Mellor S.J.; Structured Development for Real-Time Systems; Yourdon Press; 1985.
Weide94	Weide B.W., Ogden W.F.; Recasting Algorithms to Encourage Reuse; IEEE Software; setembro de 1990, pp. 80-88.
Yourdon88	Yourdon E.; Modern Structured Analysis; Yourdon Press; 1988.