Instituto de Ciências Matemáticas e de Computação

ISSN - 0103-2569

FERRAMENTAS DE ENGENHARIA REVERSA NO APOIO À QUALIDADE DE SOFTWARE

REJANE MOREIRA DA COSTA
ROSELY SANCHES

Nº 45

RELATÓRIOS TÉCNICOS DO ICMC

São Carlos
SET/1996

CAPÍTULO 1
INTRODUÇÃO

1.1. O Problema

Engenharia de Software busca estabelecer e aplicar os princípios de engenharia, objetivando produzir softwares confiáveis com baixo custo e com alta qualidade. O processo de Engenharia de Software compreende três fases genéricas: Definição, Desenvolvimento e Manutenção [PRE92].

A atividade de manutenção de software é reconhecidamente uma das fases mais problemáticas do ciclo de vida [LIE81, OSB90, DEK92, SAN93, MAN94]. Esses problemas são causadores de custos substanciais, podendo despender mais de 60% de todo o esforço de uma organização de software [LIU76, CUR79, YAU85, BEN87, SCH94]. Estima-se que mantenedores gastam entre 42 a 67 % de seu tempo tentando entender o software [OMA90].

A atividade de manutenção compreende as etapas: Entendimento, Modificação e Revalidação do software [SCH87, GAL91]. Estudos realizados por Corbi [COR89] indicam que mais da metade do tempo dos profissionais de manutenção é dedicado ao entendimento do software [COR89]. Esse autor ressalta que para modificar um software os programadores tornam-se parte historiadores, parte detetives, e parte clarividentes.

Outros estudos que buscaram identificar os fatores que fazem com que a manutenção se torne mais difícil e tão onerosa mostraram que um dos fatores principais é a inexistência ou a não completitude e/ou desatualização da documentação do software [LIE81, OSB90, DEK92, SAN93, MAN94].

Assim sendo, pode-se observar que a facilidade de manutenção (manutenibilidade), caracterizada principalmente pelo entendimento do software, está fortemente relacionada à disponibilidade de documentação sobre o software. Essa documentação pode ser obtida durante o desenvolvimento do software através de métodos, ferramentas e procedimentos de engenharia de software relacionados a cada fase do ciclo de vida [EDE94].

No entanto, na maioria das vezes a documentação é inexistente, incompleta e/ou desatualizada. A inexistência ou a desatualização da documentação são devidas, principalmente, a duas situações: software antigo, desenvolvido em uma época onde a maior preocupação tecnológica era o espaço de armazenamento e nenhum cuidado era tomado com relação às alterações efetuadas; ou o software mais recente desenvolvido utilizando algum método de engenharia de software, porém não houve preocupação com

a documentação referente a cada fase do ciclo de vida do software, principalmente com as alterações nela efetuadas [OSB90, BEN91, SAN93, WAT94, BEN95].

Nesse sentido, a Engenharia Reversa tem por objetivo principal contribuir, primeiramente, no entendimento e, posteriormente, na modificação e revalidação do software, aumentando assim a manutenibilidade do mesmo. Isto é feito através de um processo de análise que procura criar representações dos programas em uma forma mais clara ou em um nível mais alto de abstração que o código fonte [CHI90, BEN92, WAT94, STE95, SAG95].

Nos últimos anos, há um crescente reconhecimento da importância de Engenharia Reversa em ambos os campos, acadêmicos e ambientes de produção [CHI90, BEN92, WAT94, BEN95]. Este reconhecimento tem resultado na apresentação de diversas pesquisas abordando diferentes métodos, técnicas e ferramentas de engenharia reversa [BIG89, HARa90, OMAb90, BEN91, FOR92, CAN93, STA94, WAT94, PEN95, WON95].

Num ambiente mais amplo da área da Ciência de Computação, este trabalho objetiva contribuir com a apresentação de uma solução que minimize o esforço da atividade de manutenção de softwares - a engenharia reversa. Primeiramente apresentando, de forma organizada, conceitos relacionados à engenharia reversa, e em segundo lugar com o fornecimento de uma lista de ferramentas de engenharia reversa, categorizadas pela sua utilização na *visualização de código* e *entendimento de programa*.

1.2. Organização do Trabalho

Este trabalho está organizado em três Capítulos. Neste Capítulo de introdução situam-se o problema e os objetivos pretendidos.

Como o assunto deste trabalho refere-se à apresentação de uma solução que minimize o esforço de manutenção de softwares, apresentam-se no Capítulo 2 - *Manutenção de Software* - as principais características de cada fase do ciclo de vida do software, com um detalhamento maior da fase de manutenção.

Visto que uma das soluções para amenizar o esforço da atividade de manutenção de software é a aplicação de engenharia reversa para recuperação de informações, apresentam-se no Capítulo 3 - *Engenharia Reversa* - os principais conceitos, propósitos, e, finalizando, um quadro com as principais ferramentas relacionados a engenharia reversa.

CAPÍTULO 2
MANUTENÇÃO DE SOFTWARE

2.1. Considerações Iniciais

Como o assunto deste trabalho refere-se à apresentação de uma solução que minimize o esforço despendido na atividade de manutenção de software, e sabendo-se que essa atividade lida com softwares que já existem e portanto que já cumpriram as fases de um ciclo de vida, inicia-se este tópico com a colocação das principais características de cada fase do ciclo de vida do software. Como o interesse maior deste trabalho é a fase de manutenção, segue-se com um detalhamento maior de tal fase.

2.2. O Ciclo de Vida do Software

De acordo com o glossário padrão de terminologias de engenharia de software (IEEE Std 619.12-1990), desenvolvido pelo Instituto de Engenharia Elétrica e Eletrônica (IEEE) [IEE91], Ciclo de Vida de um Software é um período de tempo que se inicia quando um software-produto é concebido e que se finaliza quando o software não está mais disponível para uso.

Um ciclo de vida de software típico inclui as fases de engenharia de sistemas, análise, projeto, codificação, testes, e manutenção (Figura 2.1) [PRE92].

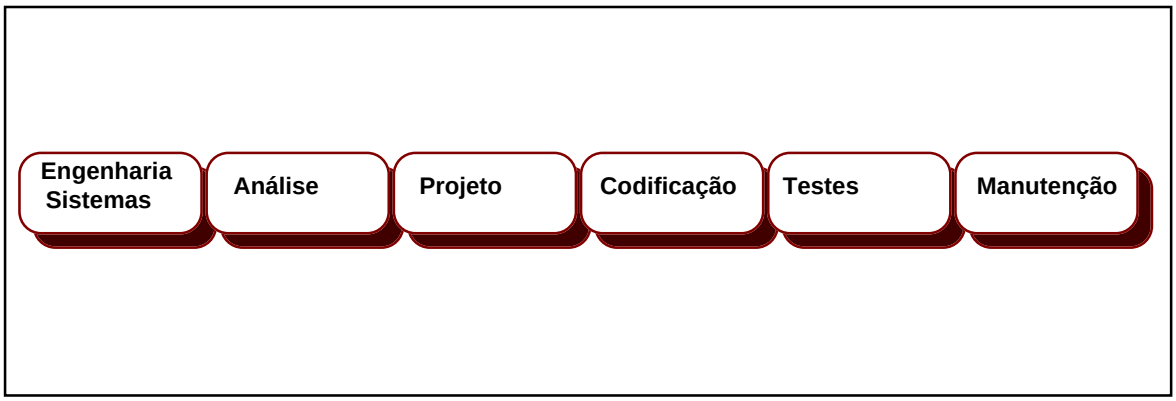

FIGURA 2.1

Ciclo de Vida Típico

Engenharia de Sistemas: Devido ao software ser sempre parte de um sistema maior, inicia-se o trabalho com o estabelecimento dos requisitos, hipóteses e restrições dos problemas.

Análise de Requisitos do Software: O domínio da informação e a função do software são focalizados mais detalhadamente. O processo de representação desse domínio pode ser visto como *especificação*.

Projeto: Os requisitos do software são traduzidos para um conjunto de representações que descrevem a arquitetura do software, as estruturas de dados e os procedimentos algorítmicos.

Codificação: As representações do projeto são traduzidas para uma linguagem de programação, resultando em instruções executáveis pelo computador.

Testes: Envolve atividades que asseguram a correta construção da lógica interna do software e a garantia de que as entradas definidas produzem os resultados esperados.

Manutenção: Após ser liberado para uso operacional o software pode necessitar de alterações. Essas alterações envolvem as fases anteriores, porém com maior complexidade e menor flexibilidade por tratar-se de um software existente.

2.3. A Fase de Manutenção de Software

Esta fase ocorre depois que o software é desenvolvido e liberado para uso operacional e envolve alterações necessárias para que ele continue sendo útil e servindo às necessidades do usuário. Algumas definições de manutenção de software são apresentadas no Quadro 2.1.

De acordo com os motivos que originam as alterações no software, a manutenção é classificada em 4 categorias [PRE92]:

1- Correção de erros não detectados durante o desenvolvimento. (Manutenção Corretiva)

2- Adequação do software a alterações de hardware ou a um novo ambiente do usuário. (Manutenção Adaptativa)

3- Atendimento a solicitação do usuário, para incluir novas capacidades, melhorar desempenho e funcionalidade. (Manutenção Evolutiva)

4- Aumento da manutenibilidade (facilidade de manutenção) do software, procurando tornar o software mais compreensível e alterável. (Manutenção Preventiva)

QUADRO 2.1

Definições de Manutenção de Software

DEFINIÇÃO	REFERÊNCIA
Modificação de um produto, após a entrega, para corrigir defeitos, para melhorar desempenho ou outros atributos, ou para adaptar o produto a um ambiente alterado.	IEEE [IEE91]
Correção, adaptação e aperfeiçoamento de software operacional.	Swanson [SWA90]
Execução das atividades necessárias para manter um software operacional e adequado aos usuários após ter sido colocado em produção.	Vallabhaneni [VAL87]
Mudanças que devem ser feitas em programas de computador, após eles terem sido entregues para o cliente ou usuário.	Martin [MAR83]
Conjunto de operações (acompanhamentos, testes, controles, ajustes, correções, adaptações e algumas otimizações e extensões) destinadas à conservação em estado operacional dos softwares computacionais em concordância com as necessidades do usuário, as operações administrativas, as exigências externas e outras influências.	Moura [MOU79]

A fase de manutenção de software é considerada uma das mais problemáticas dentre as fases do ciclo de vida do software [LIE81, OSB90, DEK92, SAN93, MAN94], sendo mesmo caracterizada como um *iceberg* [CAN72].

Estudos demonstram que essa fase pode despender mais de 60% de todo o esforço de uma organização de software [LIU76, CUR79, YAU85, BEN87, SCH94]. Uma relação de custos entre as fases do ciclo de vida do software pode ser vista na Figura 2.2 [SCH94]. Segundo Pressman [PRE92], essa proporção tende a crescer podendo atingir a faixa de 70 a 80% do orçamento global do software, nos anos 90. Visaggio [VIS94] relata um caso no qual uma organização produtora de softwares alcançou o extremo de não desenvolver novos softwares, devido ao emprego de todos os seus recursos na manutenção de sistemas antigos.

Apesar do custo monetário ser o problema mais óbvio, existem outros problemas associados à atividade de manutenção [PRE92]:

- Problemas como a perda ou adiamento de oportunidades de desenvolvimento causados pelo fato dos recursos disponíveis para essa atividade serem canalizados para tarefas de manutenção;

- A insatisfação do cliente, quando os pedidos de manutenção não podem ser mantidos em tempo hábil;

- A redução da qualidade global do sistema, como conseqüência da introdução de erros causados pelas alterações;

- Grande diminuição na produtividade dos programadores.

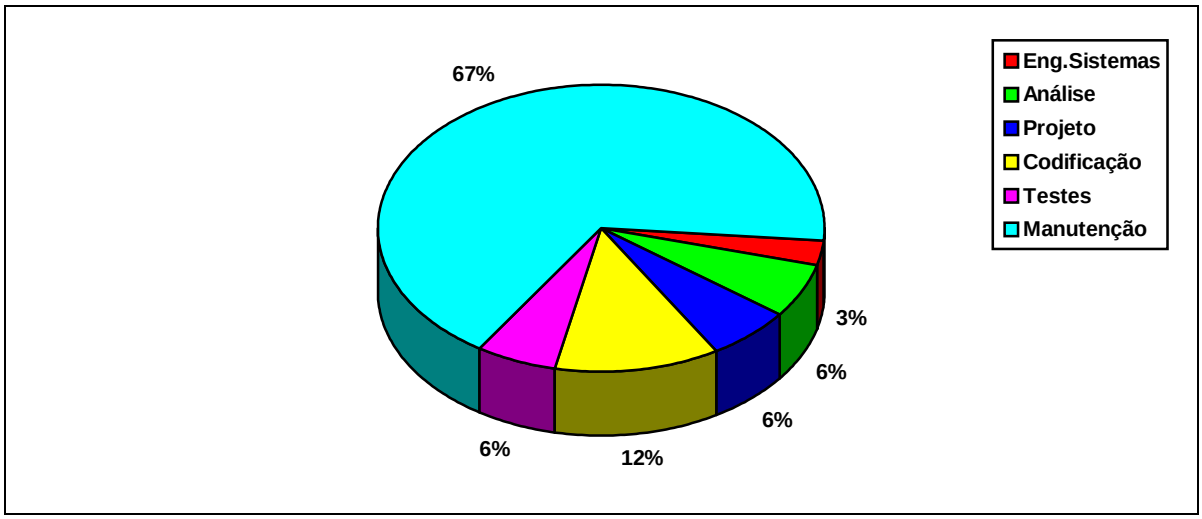

FIGURA 2.2

Relação de Custos entre as Fases do Ciclo de Vida do Software

Com o intuito de reduzir o alto esforço da atividade de manutenção, estudos têm sido realizados buscando identificar os principais problemas que ocasionam as dificuldades de desempenho dessa atividade elevando de tal forma o seu custo. Um resumo de alguns estudos é apresentado no Quadro 2.2.

Pelos estudos realizados, que buscam identificar as causas dos problemas de manutenção, constata-se que um fator relevante é a inexistência ou a não completitude e/ou desatualização da documentação de desenvolvimento e de manutenção de software.

QUADRO 2.2

Causas dos Problemas com a Manutenção de Software

CAUSAS DOS PROBLEMAS	REFERÊNCIA
- Conhecimento do Usuário - Condições do Programador - Qualidade do Produto Característica do Projeto Qualidade da Programação Original Qualidade da Documentação - Disponibilidade de Tempo - Condições da Máquina - Segurança de Sistemas	Lientz e Swanson [LIE81]
- Softwares antigos desenvolvidos com tecnologia ultrapassada e em uma época cuja preocupação era o tamanho e espaço de armazenamento - Muitas alterações (adaptações à tecnologias mais recentes, atendendo a novas necessidades dos usuários) sem considerar a arquitetura global, a documentação...	Osborne e Chikofisky [OSB90]
- Mudanças prioritárias - Métodos de teste inadequados - Documentação de sistemas incompletas ou inexistentes - Dificuldades no desempenho de medição - Adaptação do software à rápida mudança ambiental - Grande acúmulo de pedidos a executar	Dekleva [DEK92]
- Complexidade - Legibilidade - Documentação de Manutenção - Documentação de Definição - Documentação de Testes de Software - Familiaridade com o Software - Ambiente dos Dados - Uso de Ferramentas - Documentação de Projeto - Fatores Motivadores - Documentação de Codificação	Sanches [SAN93]

(Continua)

QUADRO 2.2 (Continuação)

Causas dos Problemas com a Manutenção de Software

CAUSAS DOS PROBLEMAS	REFERÊNCIA
- Documentação do software inexistente ou precária - Dificuldade de entender programas codificados por outros programadores - Programadores que deram origem ao software nem sempre estão disponíveis a responder questões sobre o mesmo. - Ausência de planejamento de modificação do software.	IEEE 1219 Working Group [MAN94]

Uma solução para superar esse problema seria o descarte do sistema atual e o re-desenvolvimento de um novo sistema no qual se teria uma preocupação maior com a documentação. Essa solução porém não é aceita na maioria dos casos, pois se reconhece o valor do acúmulo de experiências e informações obtidos durante anos e que estão embutidos no software. Além disso, muitas vezes é economicamente inviável descartar o alto investimento financeiro já atribuído ao software [BEN91, SAN93, WAT94, BEN95].

Assim sendo, outras soluções são necessárias. Uma dessas soluções é a recuperação de informações sobre o software (recuperando documentação), através da Engenharia Reversa.

2.4. Considerações Finais

Conforme foi visto neste Capítulo, a atividade de manutenção é uma das fases mais dispendiosas do ciclo de vida, e um dos principais fatores que ocasiona esse dispêndio é a inexistência ou a não completitude e/ou desatualização da documentação de software. Uma das soluções para amenizar esse esforço é a aplicação de Engenharia Reversa para recuperação de informações, a qual é discutida em maiores detalhes no Capítulo seguinte.

CAPÍTULO 3
ENGENHARIA REVERSA

3.1. Considerações Iniciais

Inicia-se este tópico com a colocação dos Níveis de Abstração no Ciclo de Vida do Software, visto que o processo de Engenharia Reversa envolve mudanças nesse nível. Neste Capítulo também apresentam-se as Visualizações, em diferentes níveis de abstração, que se pode obter do software pela Engenharia Reversa. De acordo com o nível de entendimento obtido do sistema e o escopo das informações usadas é apresentada uma Categorização de Engenharia Reversa, e finalizando este tópico, é apresentado um Quadro das Principais Ferramentas relacionadas.

3.2. Níveis de Abstração no Ciclo de Vida

As fases do ciclo de vida, apresentadas no Capítulo 2, podem ser agrupadas em três atividades fundamentais: Sistema (engenharia de sistemas), Requisitos (análise), e Desenvolvimento (projeto, codificação e testes). A fase de manutenção é vista como reiteração de atividades prévias.

A primeira atividade envolve o contexto em que o sistema está operando, ou seja o *porquê* do sistema ser desenvolvido. A segunda é descrita como o estabelecimento dos serviços a serem fornecidos pelo sistema e as restrições sob as quais ele deve operar, ou seja *o que* o sistema deve fazer e sob quais circunstâncias. A terceira é a criação de um planejamento da solução, ou seja *como* o sistema fará o proposto na atividade de requisitos, seguido da implementação da solução proposta, incluindo a codificação, os testes, a depuração, e a entrega do sistema para operação.

Para efeito desse estudo, considerar-se-ão as fases genéricas (Figura 3.1), as quais possuem uma clara diferenciação nos níveis de abstração.

Abstração é definida como a habilidade de se ignorar os aspectos de assuntos não relevantes para o propósito em questão, tornando possível uma concentração maior nos assuntos principais [OXF86]. Dois conceitos relacionam abstração com as atividades do ciclo de vida:

Nível de Abstração: Cada passo no processo de desenvolvimento de software é um refinamento do nível de abstração do software. Um alto nível de abstração tem menos detalhes que um baixo nível de abstração. Nos estágios iniciais do ciclo de vida as

informações estão representadas de uma forma mais global (possuem *alto* nível de abstração) e nos estágios finais de uma forma mais detalhada (*baixo* nível de abstração) [CHI90].

Grau de Abstração: Está relacionado a uma mesma atividade no ciclo de vida do software. Informações em uma forma mais global possuem *alto* grau de abstração, em uma forma mais detalhada, *baixo* grau de abstração (Figura 3.2) [CHI90].

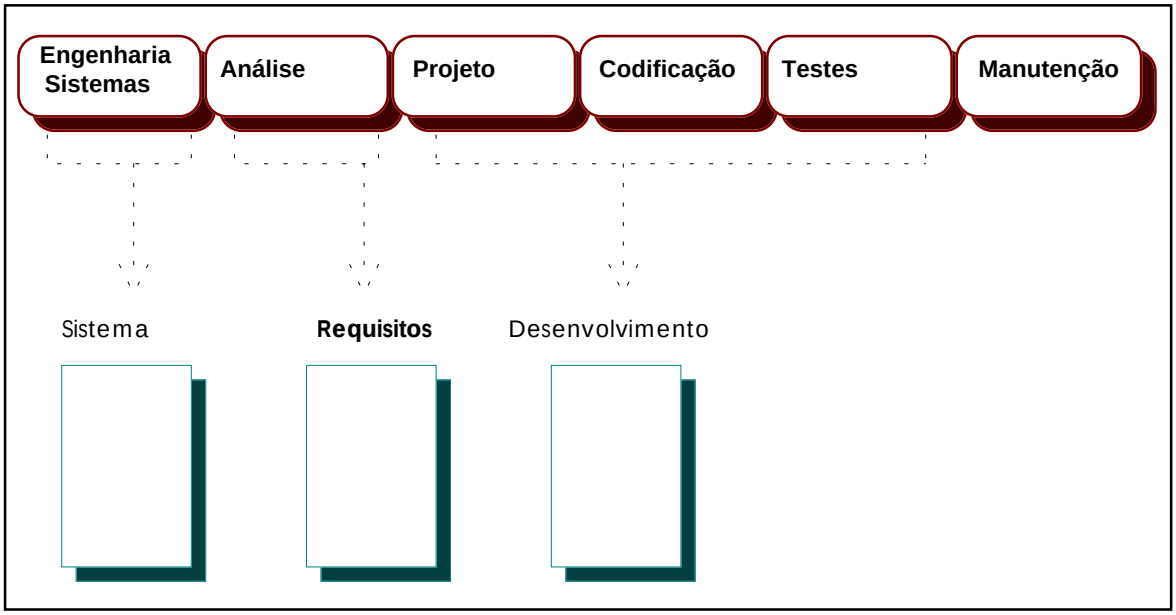

FIGURA 3.1

Atividades Fundamentais do Ciclo de Vida

O processo tradicional de engenharia de software, caracterizado pelas atividades progressivas do ciclo de vida, que partem de um alto nível de abstração, para um baixo nível de abstração, é conhecido como Engenharia Progressiva [CHI90].

O processo inverso à Engenharia Progressiva, caracterizado pelas atividades retroativas do ciclo de vida, que partem de um baixo nível de abstração para um alto nível de abstração, é conhecido como **Engenharia Reversa** (Figura 3.3) [CHI90].

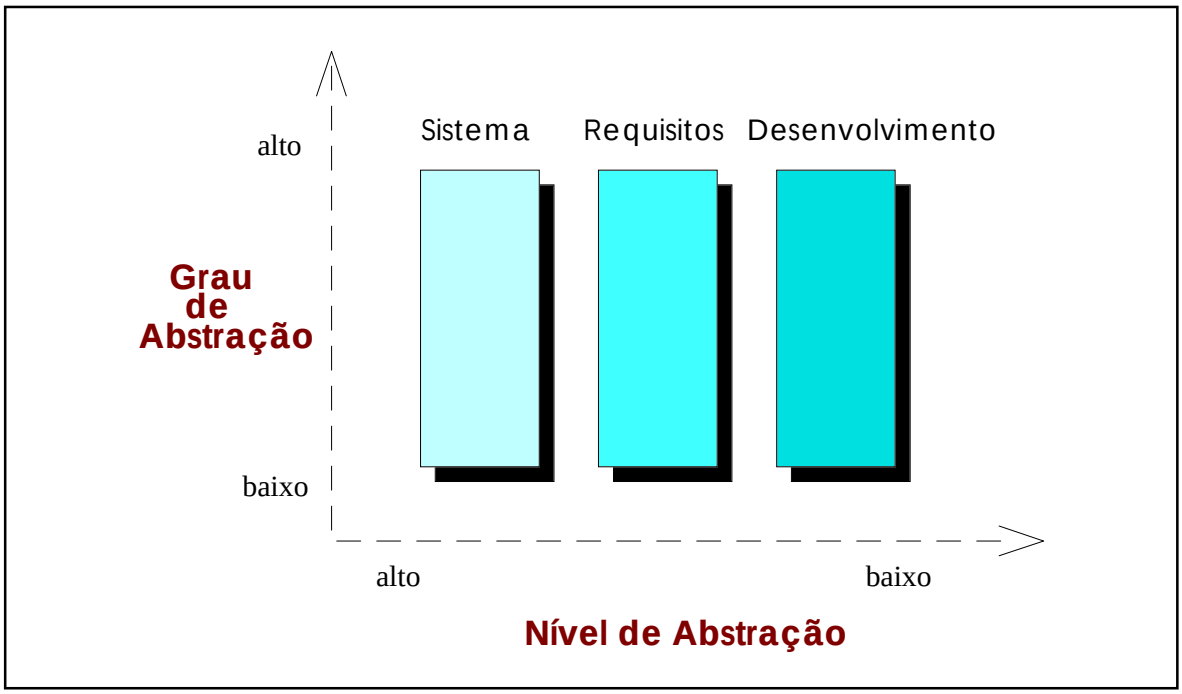

FIGURA 3.2

Níveis e Graus de Abstração no Ciclo de Vida

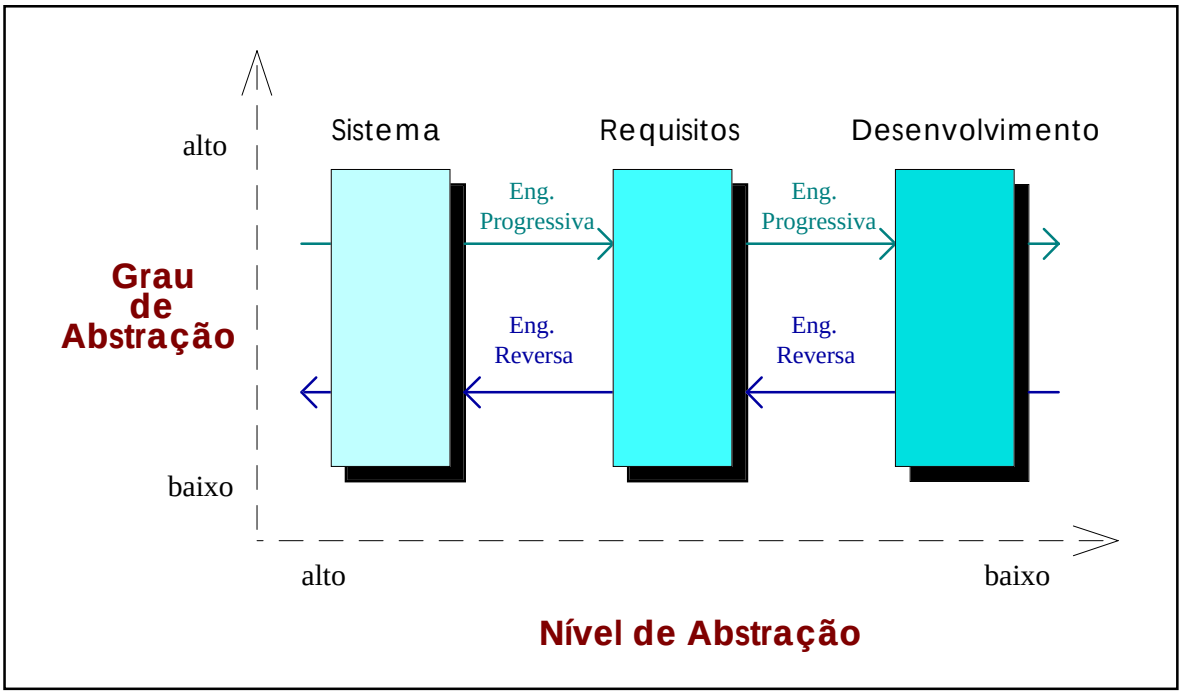

FIGURA 3.3
Relacionamento entre as atividades de Eng. Progressiva e Eng. Reversa

3.3. Definições de Engenharia Reversa

O termo " Engenharia Reversa " tem sua origem na análise de hardware, onde é comum a prática de decifrar projetos de produtos finalizados. Engenharia Reversa de Hardware é regularmente aplicada para melhorar os próprios produtos, bem como para analisar os produtos de competidores ou de adversários em situações militares ou de segurança nacional [REK85].

O conceito de Engenharia Reversa de Software é similar. Porém, enquanto tradicionalmente o objetivo da engenharia reversa de hardware é duplicar o sistema, o objetivo da engenharia de software freqüentemente é obter um entendimento do sistema em um nível maior de abstração.

O Quadro 3.1 apresenta algumas definições de engenharia reversa de software.

QUADRO 3.1

Definições de Engenharia Reversa de Software

DEFINIÇÃO	REFERÊNCIA
Processo de exame e compreensão do software existente, para recapturar ou recriar o projeto e decifrar os requisitos atualmente implementados pelo sistema, apresentando-os em um grau ou nível mais alto de abstração. Não envolve mudanças no software ou criação de um novo software	Chikofsky e Cross II [CHI90]
Processo de análise num esforço de criar uma representação do programa em um nível de abstração mais alto que o código fonte	Pressman [PRE92]
Coleção de teorias, métodos e técnicas capazes de apoiar (1) o projeto e implementação de um processo para extrair e abstrair informações de softwares existentes e produzir documentação consistente com o código, (2) a adição de conhecimentos e experiências à documentação, que não podem ser automaticamente reconstruídas a partir do código	Benedusi *et al* [BEN92]
Processo de análise de um sistema para identificar os componentes de um software e seus inter-relacionamentos, e para criar uma representação do software em outra forma, provavelmente num nível mais alto de abstração que o código fonte	Waters e Chikofsky [WAT94]
Processo de engenharia para entender, analisar, e abstrair o sistema para uma nova forma, num alto nível de abstração	Stephen e Lynn [STE95]
Processo através do qual um dado sistema ou produto é examinado para identificar ou especificar a definição do produto em nível de sistemas, em nível de requisitos, ou em nível de projeto	Sage [SAG95]

Através da engenharia reversa um software pode ser visualizado em diferentes níveis de abstração. Cada visualização abstrai características próprias da fase do ciclo de vida correspondente à abstração.

3.4. Visualizações do Software

Segundo Harandi e Ning [HARa90] um software pode ser visualizado a partir de diferentes níveis de entendimento. Baseado nos níveis de abstração, as visões são classificadas em 4 *categorias*: visão em nível-implementacional, visão em nível-estrutural, visão em nível-funcional, visão em nível-de-domínio.

A Figura 3.4 mostra a correspondência entre as categorias de visualização do software e as informações produzidas e utilizadas nas diferentes atividades do ciclo de vida do software.

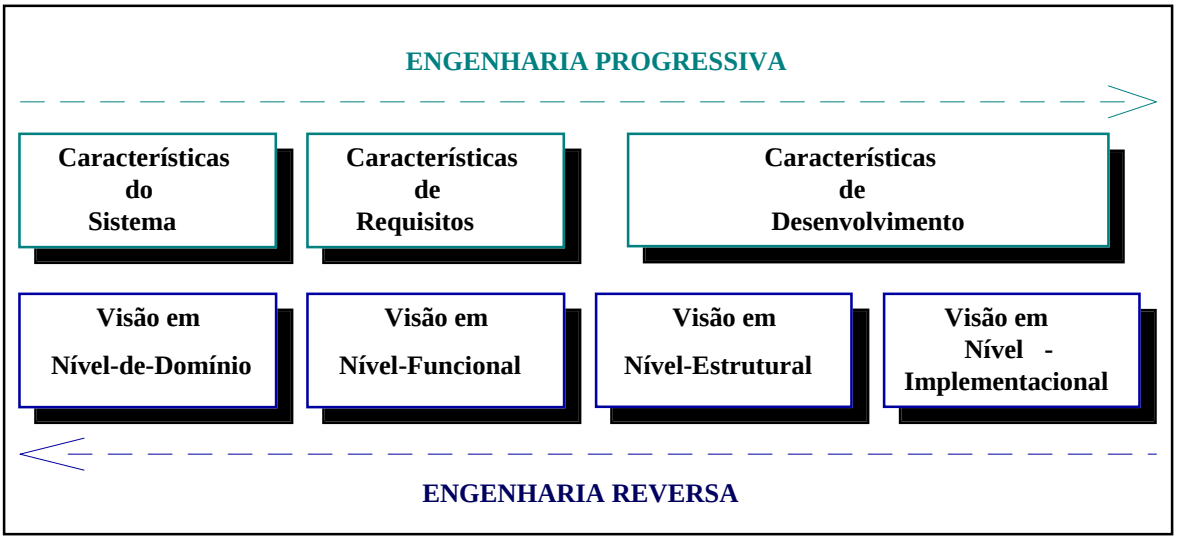

FIGURA 3.4
Níveis de Entendimento do Software de acordo com o Ciclo de Vida

Visão em nível-implementacional Abstrai características da linguagem de programação e características específicas da implementação. Exemplos de visões em nível implementacional são informações a respeito da sintaxe e da semântica da linguagem e informações da implementação.

Visão em nível-estrutural Abstrai detalhes da linguagem de programação para revelar sua estrutura a partir de diferentes perspectivas. O resultado é uma representação

explícita das dependências entre os componentes do sistema. Exemplos de visões em nível estrutural são grafos de fluxo de dados, grafos de fluxo de controle [CLE89], o projeto procedimental expresso através de uma linguagem de projeto de programação [CAI75, HARb90], o projeto arquitetural expresso através de gráficos de estruturas [CRO90], ou através de uma linguagem de interconexção modular, como NuMIL [CHO90].

Visão em nível-funcional Abstrai a função de um componente, isto é, *o que* o componente faz. Essa visão relaciona partes do programa às suas funções procurando revelar as relações lógicas entre elas (diferentemente das relações sintáticas ou da estruturais). Exemplos de visões em nível funcional podem ser descrições da função do sistema expressos de maneira formal usando linguagens tais como VDM [JON90], Z e Z^{++} [SPI88, KHA89, BEN91]; diagramas de fluxo de dados [DEM79], que apresentam os processos e o fluxo de dados entre eles; diagramas de fluxo de controle [HAT87], que apresentam os processos e o fluxo de controle entre eles e diagramas de entidade-relacionamento [ROS88], que mostram os dados e seus relacionamentos.

Visão em nível-de-domínio Abstrai o contexto em que o sistema está operando, ou seja o *porquê* do sistema a ser desenvolvido.

Na realidade poucas representações são restritas somente a uma fase do ciclo de vida ou consideradas como pertencentes a uma categoria de visualização. Uma linguagem de projeto de programa [CAI75] pode ser usada para representar as funções na fase de requisitos ou o projeto procedimental na fase de projeto. Um diagrama de fluxo de dados [DEM79], pode ser usado para descrever o que o sistema faz ou como o processo interage. Assim, se uma representação extraída é uma representação de requisitos ou uma representação de projeto depende principalmente do contexto em que a representação será usada.

É relevante ressaltar que uma forma de representação extraída do código pode diferir de uma representação similar que foi desenvolvida no processo de engenharia progressiva. A forma extraída irá refletir a idiossincrasia da representação do código muito mais do que a representação original, que reflete a compreensão do problema pelo analista ou projetista.

Para se obter as diversas "visões" do software, muitas vezes é necessário acrescentar às informações contidas no código outras informações provenientes de conhecimentos e experiências humanas. De acordo com o nível de entendimento obtido do sistema e o escopo das informações usadas pode ser formulada uma categorização das técnicas de engenharia reversa.

3.5. Categorias de Engenharia Reversa

De acordo com o nível de entendimento obtido do sistema e o escopo das informações usadas, duas categorias de engenharia reversa são definidas: Visualização de Código [OMAa90] e Entendimento de Programa [CHI90].

Na Figura 3.5, são apresentadas as categorias de engenharia reversa (Visualização de Código e Entendimento de Programa), relacionadas às fases do ciclo de vida.

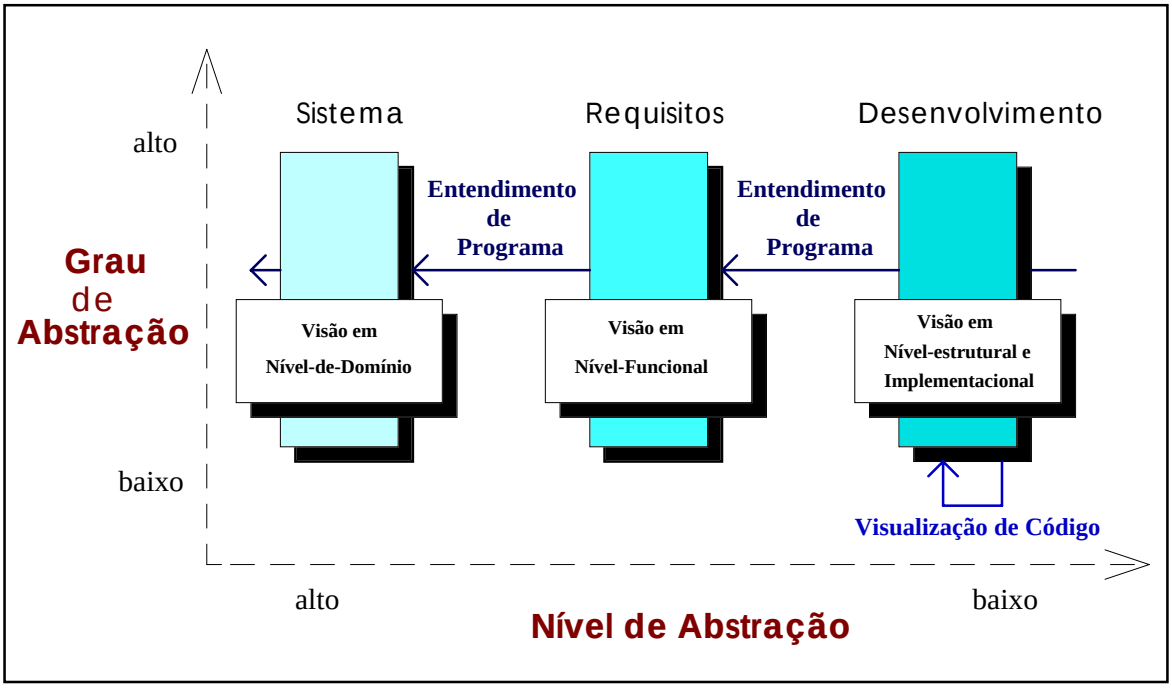

FIGURA 3.5

Categorias da Engenharia Reversa relacionadas ao Ciclo de Vida

Visualização de Código **:** Também denominada *Redocumentação* [CHI90]. É a criação ou revisão de representações semanticamente equivalentes num mesmo nível de abstração [CHI90]. O processo de Visualização de Código cria as representações a partir de informações obtidas apenas da análise do código fonte, embora a apresentação dessas informações possa se diversificar. As formas das representações são consideradas visões alternativas, cujo o objetivo é melhorar a compreensibilidade do sistema global.

Visualização de Código é a forma mais simples e mais antiga de engenharia reversa. A intenção é recuperar a documentação que já existiu, ou que deveria ter existido, sobre o sistema. A ênfase, de fato, é a criação de visões adicionais, especialmente visões gráficas, que não foram criadas durante o processo original de engenharia progressiva.

As ferramentas mais comuns usadas para desempenhar a Visualização de Código são apresentadas no Quadro 3.2. (pag.24). As ferramentas usam o código fonte do software como entrada, analisam e extraem a arquitetura do programa, a estrutura de controle, o fluxo lógico, a estrutura de dados, o fluxo de dados e o fluxo de controle. Algumas ferramentas dessa categoria aplicam uma técnica denominada Fatiamento de Programa (*Program Slicing*) [GAL91]. Nessa técnica, o mantenedor especifica os tipos de estruturas de programa (declarações de dados, laços) que são de interesse e a ferramenta de engenharia reversa remove o código estranho, possibilitando que somente o código de interesse seja representado, para análise dos efeitos produzidos pelas mudanças. Outras ferramentas aplicam a técnica *Análise de Dependência* [WIL90], construindo mapas gráficos de dependências que mostram as ligações entre as estruturas de dados, e entre os componentes do programa.

O objetivo das ferramentas de Visualização de Código é fornecer meios fáceis para visualizar o relacionamento entre os componentes do programa, facilitando a compreensibilidade do sistema. Essas ferramentas simplesmente auxiliam o entendimento do sistema. Através delas não se obtém informações das funções ou propostas do sistema.

Esse nível de entendimento não transcede a visão em *nível-estrutural* e não atribui significados para o sistema analisado. Recuperações mais ambiciosas tais como a função, os propósitos ou a essência do sistema, exigem um nível de entendimento maior e são definidas como *Entendimento de Programa*.

Entendimento de Programa: Nesta categoria de engenharia reversa, também denominada *Recuperação de Projeto* [CHI90], o conhecimento do domínio das informações externas e as deduções são adicionadas às observações feitas sobre o sistema através do exame do mesmo, de modo a obter informações com nível mais alto de abstração [CHI90].

Segundo Biggerstaff [BIG89], *Entendimento de Programa* recria abstrações do projeto a partir de uma combinação de código, documentação existente do projeto (se disponível), experiências pessoais, e conhecimentos gerais sobre o problema e o domínio de aplicação. Sintetizando, Entendimento de Programa deve produzir todas as

informações necessárias para se entender completamente *o que* o sistema faz, *como*, e *por que* o sistema faz.

As ferramentas usadas para desempenhar Entendimento de Programa são apresentadas no Quadro 3.3. Muitas das ferramentas de Entendimento de Programa possuem alguma forma de base de conhecimento que une *padrões de programação* a *conceitos funcionais*. Reconhecimento de Padrões (*Pattern Matching*) [BIG89] é um mapeamento de construções de baixo nível para conceitos de alto nível. Um modo de se fazer esse mapeamento é através de uma base de conhecimentos constituída de modelos de estrutura de programação para interpretação do código a ser analisado. Outro modo de fazer o mapeamento é através do emprego da técnica Análise Baseada na Intenção (*Intention-Based Analysis*) [JOH90]. Nessa técnica, a base de conhecimentos é constituída de descrições de como os programadores escrevem os programas (intenções dos programadores), formando algoritmos hipotéticos. As descrições da base de conhecimento são pareadas com descrições do código analisado para detectar possíveis ocorrências de erros.

As ferramentas de Entendimento de Programa distinguem-se das ferramentas de Visualização de Código porque objetivam *entender* o sistema, em vez de simplesmente fornecer visões alternativas para *auxiliar* o usuário a entender o sistema. Esse entendimento vai além do conhecimento em *nível implementacional e estrutural*, buscando obter o conhecimento em *nível-funcional* e até mesmo em *nível-de-domínio* (ambiente de operação do sistema).

Um completo Entendimento de Programa busca reconstruir não somente a função do sistema, mas também o processo pelo qual o sistema foi desenvolvido. Rugaber *et al.* [RUG90] enfatizam a importância da recuperação de decisões de projeto tomadas durante o desenvolvimento original para uma completa estrutura de entendimento.

Entendimento de Programa é a forma mais crítica de engenharia reversa porque tenta imitar o raciocínio humano na busca do entendimento.

Na Figura 3.6, são apresentadas as categorias de engenharia reversa (Visualização de Código e Entendimento de Programa), definidas pelas visões que elas obtém e o escopo das informações usadas, seja pela análise pura do código ou pela análise do código combinada com a tecnologia de base de conhecimento.

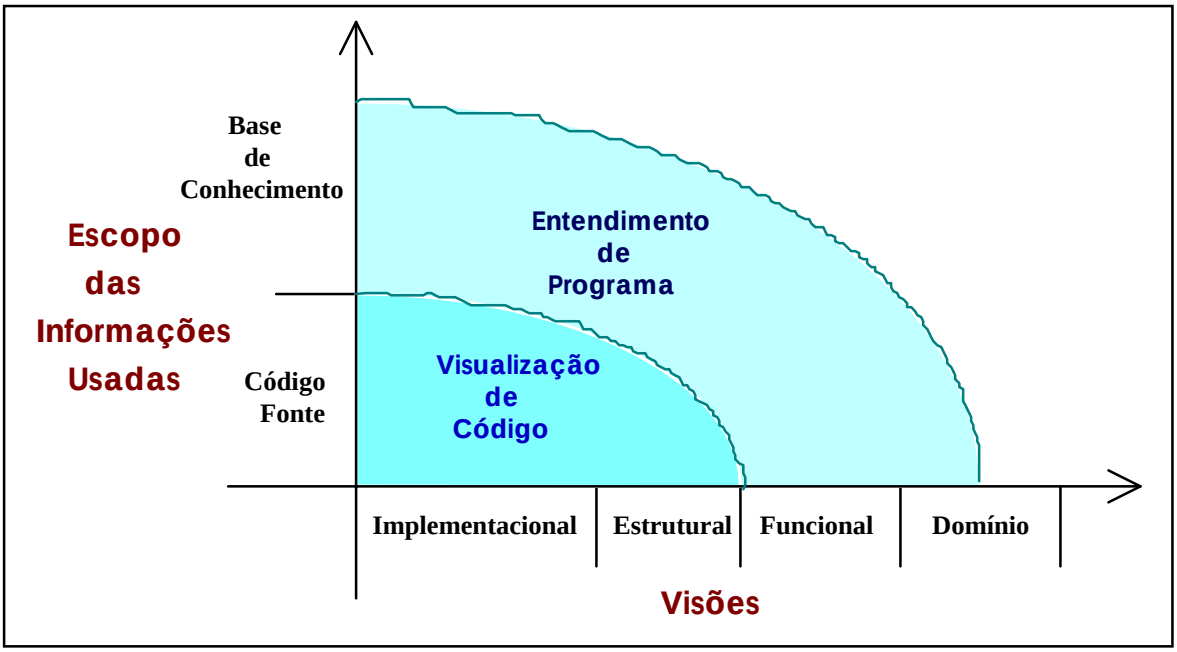

FIGURA 3.6

Características Fundamentais de

Visualização de Código e Entendimento de Programa

O *entendimento de programa* e a *visualização de código* permitem que o software seja visualizado em diferentes níveis de abstração. No entanto, para serem operacionalizadas efetivamente, essas categorias de engenharia reversa necessitam de apoio de ferramentas.

3.6. Ferramentas de Engenharia Reversa

Nesta seção, apresentam-se as ferramentas de engenharia reversa categorizadas pela sua utilização no *entendimento de programa* e a na *visualização de código.* No quadro das ferramentas de visualização de código identifica-se a ferramenta, a linguagem e a plataforma às quais ela se aplica, a visualização produzida e referências nas quais maiores detalhes sobre a ferramenta podem ser obtidos (Quadro 3.2). No quadro das ferramentas de entendimento de programa, identifica-se a ferramenta, a linguagem à qual ela se adequa, a base de conhecimentos necessários, a estratégia de entendimento e referências nas quais maiores detalhes podem ser obtidos (Quadro 3.3).

QUADRO 3.2
Ferramentas de Visualização de Código

Nome da Ferramenta	Linguagens e Plataformas	Visualização produzida pela Engenharia Reversa	Grupo ou Companhia; Referências
ADAGEN	Ada	- Conjunto hierárquico de Diagramas Buhr, apresentando a topologia do sistema, procedimentos, funções, e chamadas de tarefas - Gráficos de dependências do sistema	Rozenblat e Fischer [ROZ89]
AD/Advantage	Aplicativos de banco de dados Interface: McCabe & Associates Inc., e Centerline Software Inc., e Code Center	-Diagramas dos relacionamentos lógicos e de dados	Cincom Systems Inc.; Stapleton [STA94]
BACHMAN TOOL SET	Cobol e Bancos de Dados Plataformas IBM PS/2 e compatíveis, MS-DOS e PCs. Sistema Operacional: MS-DOS e OS/2	- Diagramas de entidade-relacionamento a partir de bancos de dados IMS e descrições Cobol	Advanced Software Automation Inc.; Forte [FOR92]
BATTLE/MAP ACT	C, Cobol, Fortran, Ada, Basic, PL/I, Pascal PDLs... PCs sob MS-DOS, Sun workstations, DEC VAX workstations sob Unix e DEC VAX mainframes sob VMS com Ultrix	-Diagrama hierárquico onde cada módulo é colorido de acordo com a complexidade (ciclomática e essencial) * Apoio a produção de caminhos de teste para teste de integração e unidade	McCabe & Associates; McCabe [MCC90]
BOOK-MAKER	C, Pascal	Formatação do código em forma de livro (*Book Paradigm)*. - Gráfico de estruturas do programa como Tabelas de conteúdos - Módulos como Capítulos - Declarações de programação (*if,case*) como Parágrafos...	University of Idaho-Oregon State University; Oman e Cook [OMAb90]
CDT **C**oncurrent Data Flow **D**iagrams **T**ool	ADA Sun Workstation	-Diagrama de fluxo de dados para ambientes concorrentes	DIS (Dipartmento di Informatica e Sistematica) of The University of Naples Canfora [CAN93]

(Continua)

QUADRO 3.2

Ferramentas de Visualização de Código (Continuação)

Nome da Ferramenta	Linguagens e Plataformas	Visualização produzida pela Engenharia Reversa	Grupo ou Companhia; Referências
DEPENDENCY ANALYSIS TOOL SET	C PCs sob MS-DOS e workstations - Unix	- Visualização e Análise de dependências (de dados, de chamadas, funcionais) em linguagens C (seleção de tipos de dependências, dependências indiretas e falsas dependências)	University of West Florida; Wilde [WIL90]
EDSA **E**XPERT **D**ATA FLOW e **S**TATIC **A**NALYSIS TOOL	Ada PCs sob MS-DOS, Sun, Apolo, DEC workstations sob Unix, e VAX mainframes sob Unix	- Fluxo de dados - Fluxo de controle - Referências-cruzadas - Visões de perspectivas do código (*Técnica Slicing)*	Array Systems Computing; Vanek e Davis [VAN90]
ENSEMBLE	C, C^{++}, Fortran Principais plataformas UNIX, e PCs	-Diagramas de fluxo de dados - Análise de dependência de dados -Dicionário de dados -Especificações de módulos -Métricas de complexidade	Cadre Technologies Inc; Stapleton [STA94]
GRASP/Ada	Ada Sun workstations sob SunOs 4.0.3 e X Windows 11.7	- Diagramas de Estruturas de Controle (extensão do diagrama de fluxo de dados, gráficos de dados) - Diagramas de objetos (a partir de PDL/Ada ou código fonte)	Auburn University e participação da Marshall Space Flight Center-NASA; Cross [CRO90]
HINDSIGHT	C, C^{++}, Fortran Principais plataformas UNIX, e possui uma versão para PC	- Gráficos de estruturas e diagramas para geração de relatórios	Advanced Software Automation Inc.; Stapleton [STA94]
LOGISCOPE (anteriormente VERILOG INC.)	C, C^{++}, Fortran, Cobol, Pascal... Principais plataformas UNIX, e sistema operacional MVS da IBM	-Análise e informações da complexidade do código através de métricas e gráficos de chamadas. -Checagem a conformidade de padrões de programação	Logiscope Tecnologies Inc.; Stapleton [STA94]
OBJECTIVE-C BROWSER	Objective-C Sun 3,4 e 386i, HP 9000, DEC VAX sob Unix	- Hierarquia de herança de classes - Referências-cruzadas de dados - Estatística do uso de entidades definidas pelo código	Stepstone; Novobilski [NOV90]

(Continua)

QUADRO 3.2

Ferramentas de Visualização de Código (Continuação)

Nome da Ferramenta	Linguagens e Plataformas	Visualização produzida pela Engenharia Reversa	Grupo ou Companhia; Referências
PUNS **P**rogram **Un**derstanding **S**upport Environment	IBM Systems / 370 Assembler Language	- Fluxo de dados - Fluxo de controle - Interface do usuário, que permite navegar através de objetos de interesse (módulos e variáveis)	IBM's T.J. Watson Research Center; Cleveland [CLE89]
REVOLVE		- Informações dos componentes do sistema existente, extraídas de pesquisas SQL de um Banco de Dados	Burl Software Inc.; Stapleton [STA94]
SEELA	Ada, Cobol, C, Pascal, PL/M, Fortran DEC VAX / VMS mainframes e workstations	- Linguagem de projeto de programa a partir do código fonte, com edição gráfica e documentação da estrutura do código com: listagens *top down* e sequênciais, diretórios de blocos e índices de definição de módulos	Tuval Software Industries; Harband [HARb90]
SmartSystem	C, Cobol DEC VAX, Sun 3/4 Interface: Cadre's teamwork tools	- Grafos de chamadas e árvores de dependências de dados	Procase; Forte [FOR92]
STP **S**oftware **T**hrough **P**ictures		- Diagramas gráficos do código com descrição dos principais elementos do banco de dados, funções ou subrotinas que chamam, que são chamadas, variáveis usadas e escopo	Interactive Development Environment Inc.; Stapleton [STA94]
SURGEON´S ASSISTANT	C Sun workstations com Sun View sob SunOS version 4.0	Fatias (*slices)* de programas para trilhar os efeitos das estruturas modificadas (*Técnica Slicing*)	Loyola College; Gallagher [GAL90]
VIA / SMARTDOC	Cobol Plataformas IBM S/3XX e sistema operacional MVS Interface: IBM's Language Environment / 370 e Cobol / 370	- Documentação Cobol, com informações sobre fluxo de dados e lógica, e tabela de conteúdos, índices principais, onde todos os arquivos são definidos, chamados, e modificados	Viasoft; Forte [FOR92]
VIFOR **VI**EW **FOR**MS	Fortran Sun workstations sob Sun News, DEC VAXstation 2000 e MicroVAX IIGPSs sob Ultrix	- Transformações do código para forma visual e vice-versa -Visões gráficas da hierarquia de chamadas através: 1-Processos (Programas, subrotinas e funções) 2-Variáveis comuns	Software Tools and Technologies; Rajlich [RAJ90]

QUADRO 3.3
Ferramentas de Entendimento de Programa

Nome da Ferramenta	Linguagens e outros conhecimentos	Estratégia de Entendimento	Grupo ou Companhia; Referências
DESIRE **Des**ign **I**nformation **R**ecovery **E**nvironment Protótipo	C Modelo de Domínio (base de conhecimentos)	*Pattern Matching*; *Abstrações conceituais*, que são informações formais e informais conhecidas, transformadas em *idiomas* usados para localizar padrões na implementação	MCC Microelectronics and Computer Technologie Corporation Biggerstaff [BIG89] [BIG94]
PAT **P**rogram **A**nalysis **T**ool Protótipo	Pascal Base de planos (regras de inferência) em evolução	*Pattern Matching*; Reconhecimento de conceitos baseados em heurísticas, para obter conceitos funcionais do código	University of Illinois (Urbana-Champaign), Arthur Andersen and Company Harandi e Ning [HARa90]
RECOGNIZER (Do projeto Programmer´s Apprentice) Protótipo	Common Lisp Programas traduzidos em grafos de fluxo	*Pattern Matching*; Reconhecimento de todas as ocorrências de *clichês* (estruturas comuns de programação), produzindo uma descrição hierárquica do programa em termos dos *clichês* encontrados	MIT Massachusetts Institute of Technology Rich e Wills [RIC90]
PUDSY Protótipo	Pascal Programa decomposto em *chunks* e unidos a base de conhecimentos	*Pattern Matching*; Sistema de depuração *Chunks* (pedaços) do código, transformados em declarações de cálculos, que são testadas por equivalência com a especificação do programa, fornecendo soluções para as não-equivalências	F.J. Lukey Seviora [SEV87]
FUNCTION ABSTRACTION (nome da técnica) Sem protótipo	Cobol Programa estruturado (usado com Ferramenta de reestruturação IBM Cobol Structuring Facility)	Aplicação de conceitos funcionais, combinados com análise de dados, *slicing, e Pattern Matching,* a partir de algorítmos algébricos	IBM Systems Integration/ University of Maryland Hausler *et al* [HAU90]
LAURA Protótipo	Fortran Programa "correto"	*Intention-based Analysis* Comparação de um programa "correto" com o programa objeto, ambos traduzidos em grafo de fluxos normalizados	A. Adam e J.P. Laurent Seviora [SEV87]
TAULUS Protótipo	Lisp "Solução ideal"	*Intention-based Analysis* Comparação da solução ideal com um programa objeto, através de algorítmos hipotéticos	W.R. Murray of University of Texas Ourston [OUR89]

(Continua)

QUADRO 3.3

Ferramentas de Entendimento de Programa (Continuação)

Nome da Ferramenta	**Linguagens e outros conhecimentos**	**Estratégia de Entendimento**	**Grupo ou Companhia; Referências**
PROUST Protótipo	Pascal Regras-de-diferentes-planos Base de conhecimento (em evolução)	*Intention-based Analysis* Reconhecimento de erros baseados em intenções, através de algorítmos hipotéticos Fornece explicações das diferenças (baseadas nas regras)	University of Southern California Johnson [JOH90]
RIGI	Cobol, C, Unix	Redocumentação estrutural automática, produzindo grafos de fluxo de recursos e Reconhecimento de padrões e técnicas de composição de sistema para geração de abstrações de alto nível	University of Victoria Wong *et al* [WON95]
REDO	Cobol, Fortran	Reestruturação, incluindo estruturas de dados, variáveis locais, e estruturas de controle e Redocumentação a partir do código e uma LI (Linguagem Intermediária - UNIFORM), para uma linguagem de especificação formal (Z e Z^{++}) Armazenamento das informações extraídas em um banco de dados	ESPRIT II Collaborative Programme of Research; Khabaza[KHA89], Bennett[BEN91], Jonathan[JON93]
REFORM	IBM Assembler * futuramente Cobol - Biblioteca de transformações - Base de Conhecimentos	Produção de especificação a partir do código via métodos formais Tradução do código em WSL (Wide Spectrum Language)	Centre for Software Maintenance at Durham University in Collaboration with IBM (UK) Ltd Bennett [BEN91]
MACS	C * futuramente Cobol - Repositório com conhecimentos do domínio de aplicação e implementação, e das perícias do mantenedor	Abordagem de assistência através de um sistema especialista Extração do projeto e estrutura do código, e representação em um formalismo independente da linguagem (*dimensional design*)	ESPRIT II Collaborative Programme of Research Bennett [BEN91]

3.9. Considerações Finais

Neste Capítulo foram abordados os principais conceitos relacionados à Engenharia Reversa de Software. Diferentes métodos, técnicas e ferramentas de engenharia reversa foram apresentados. Foi apresentada uma lista de ferramentas de engenharia reversa, categorizadas pela sua utilização na *visualização de código* e *entendimento de programa.*

REFERÊNCIAS BIBLIOGRÁFICAS

[ALB79] ALBRECHT, A.J. Measuring Application Development Productivity. In: *Proc. IBM Application Deveopment Symposium.* Monterey, C.A., p.83-92, 1979

[ARN92] ARNOLD, R.S.; FRAKES, W.B. Software Reuse and Reengineering. CASE Trends, v.4, n.2, p.44-8, 1992

[BEN87] BENDIFALLAH, S.; SCACCHI, W. Understanding software Maintenance Work. IEEE Transaction on Software Engineering, v.13, n.3, p.311-23, 1987

[BEN91] BENNET, K.H. Automated Support of Software Maintenance. Information and Software Technology, v.33, n.1, p.74-85, 1991

[BEN92] BENEDUSI, P.; CIMITILE, A.; CARLINI, U. Reverse Engineering Processes, Design Document Production, and Structure Charts. Journal Systems and Software, v.19, p.225-45, 1992

[BEN95] BENNET, K.H. Legacy Systems. IEEE Software, v.12, n.1, p.19-23, 1995

[BIG89] BIGGERSTAFF, T.J. Design Recovery for Maintenance and Reuse. IEEE Computer, v.22, n.7, p.36-49, 1989

[BIG94] BIGGERSTAFF, T.J. et al. Program Understanding and the Concept Assignment Problem. Communications of the ACM, v.37, n.5, 1994

[BOL89] BOLDYREFF, C. Reuse, Software Concepts, Descriptive Methods and the Practitioner Project. ACM SIGSOFT Software Engineering Notes, v.14, n.2, 1989

[BOO91] BOOCH, G. Object-Oriented Design with Applications. Benjamin Cummings. CA. 1991

[CAI75] CAINE, S.; GORDON K. PDL- A Tool for Software Design. In: Proc. National Computer Conference. AFIPS Press, p.271-6, 1975

[CAN72] CANNING, R. The Maintenance "Iceberg". EDP Analyser, v.10, n.10, 1972

[CAN93] CANFORA, G. et al. A Reverse Engineering Process for Design Level Document Production from ADA Code. Information and Software Technology, v.35, n.1, p. 23-34, 1993

[CAN94] CANFORA, G.; CIMITILE, A.; MUNRO, M. RE2 : Reverse Engineering and Reuse Reengenharia. Journal Software Maintenance : Research and Practice, v.6, p.53-72, 1994

[CHI90] CHIKOFSKY, E.J.; CROSS II, J.H. Reverse Engineering and Design Recovery: A Taxonomy. IEEE Software, v.7, n.1, p.13-7, 1990

[CHO90] CHOI, S.C.; SCACCHI, W. Extracting and Reestructuring the Design of Systems. IEEE Software, v.7, n.1, p.66-71, 1990

[CLE89] CLEVELAND, L. A Program Understanding Support Environment. IBM Systems Journal, v.28, n.2, p.324-44, 1989

[COR89] CORBI, T.A. Program Understanding: Challeng for the 1990s. IBM Systems Journal. v.28, n.2, 1989

[CRO90] CROSS, J.H. Grasp/Ada Uses Control Structure. IEEE Software, v.7, n.3, p.62, 1990

[CUR79] CURTIS, B. et al. Measuring the Psychological Complexity of Software Maintenance Tasks with the Halsted and McCabe Metrics. IEEE Conference on Software Engineering, v.5, n.2, p.96-104, 1979

[DEM79] DeMARCO, T.; SARSON, C. Structured Analysis and Systems Specification. Prentice-Hall, 1979

[DEK92] DEKLEVA, S. Delphi Study of Software Maintenance Problems. In: Proc. of 8th International Conference on Software Maintenance, 1992

[EDE94] EDELSTEIN, D.V. Standard for Software Maintenance - Report on the IEEE STD 1219-1993, Software Engineering Notes, v.18, n.8, p. 94-5, 1994

[FOR92] FORTE, G. Tools Fair: Out of the lab, onto the shelf - Reverse Engineering and Maintenance. IEEE Software, V.9, N.3, p.76, 1992

[FRA92] FRAZER, A. Reverse Engineering - Hipe, Hope or Here? In: HALL, P.A.V. - Software Reuse and Reverse Engineering in Practice. London, Chapman & Hall, 1992

[GAL90] GALLAGER, K.B. Surgeon´s Assistant Limits Side Effects. IEEE Software, v.7, n.3, p.64, 1990

[GAL91] GALLAGER, K.B.; LYLE, J.R. Using Program Slicing in Software Maintenance. IEEE Transactions on Software Engineering, v.17, n.8, p.751-61, 1991

[HAT87] HATLEY, D.J.; PIRBHAI, I.A. Strategies for Real Time Systems Specifications. Dorset House, 1987

[HARa90] HARANDI, M.T.; NING, J.Q. Knowledge-Based Program Analysis. IEEE Software, v.7, n.1, p.74-81, 1990

[HARb90] HARBAND, J. Seela Aids Maintenance with Code-Block Focus. IEEE Software, v.7, n.3, p.61, 1990

[HAU90] HAUSLER, P.A. Using Function Abstraction to Understand Program Behavior. IEEE Software, v.7, n.1, p.55-63, 1990

[IEE91] IEEE Standard Glossary of Terminology in Software Engineering. In: IEEE Software Engineering Standard Collection, p.07-83, 1991

[JOH90] JOHNSON, W.L. Understanding and Debugging Novice Programs. Artifial Intelligent, v.42, n.1, p.51-97, 1990

[JON90] JONES, C.B. Systematic Software Development Using VDM. 2.ed. s.l., Prentice-Hall, 1990

[JON93] JONATHAN, B.; BREUER, P.; LANO, K. A Compendium of Formal Techniques for Software Maintenance. Software Engineering Journal, v.8, n.5, p.253-62, 1993

[KHA89] KHABAZA, I. Maintenance, Validation, and Documention of software Systems: 'REDO'-ESPRIT P2487. In CASE '89: Proc. of the Third International Workshop on Computer-Aided Software Engineering, British Computer Society, p.221-2, 1989

[LIE81] LIENTZ, B.P.; SWANSON, E.B. Problems in Application Software Maintenance. Comunications of ACM, v.24, n.11, p.31-7, 1981

[LIU76] LIU, C.C. A Look at Software Maintenance. Datamation, nov., p.51-5, 1976

[MAN94] MAMONE, S. The IEEE Standard for Software Maintenance. Software Engineering Notes, v.19, n.1,1994

[MAR83] MARTIN, J.; McCLURE, C. Software Maintenance: The Problem and Its solutions. Englewood Cliffs, NJ : Prentice-Hall, 1983

[MAS91] MASIERO, P.C.; FORTES, R.P.M.; BATISTA, J.E.S. Edição e Simulação de aspectos Comportamentais de Sistemas de tempo Real. In: XVIII Seminário Integrado de Software e Hardware, Santos, Brazil, 1991

[MAS94] MASIERO, P.C.; MALDONADO, J.C.; BOAVENTURA, I.A.G. A Reachability Tree for Statecharts and Analysis of Some Properties. Information and Software Technology, v.36, n.10, 1994

[MCC90] McCABE, T.Jr. Battle-Map, ACT Show Code Structura, Complexit. IEEE Software, v.7, n.3, p.62, 1990

[MOU79] MOURA, E.T. Manutenção de Software . Dissertação (Mestrado) - Departamento de Informática, Pontifícia Universidade Católica, Rio de Janeiro, 1979

[NOV90] NOVOBILSKI, A. Objective-c Browser Details Class Structure. IEEE Software, v.7, n.3, p.60, 1990

[OMAa90] OMAN, P.W. Maintenance Tools. IEEE Software, v.7, n.3, p.59-65, 1990

[OMAb90] OMAN, P.W.; COOK, C.R. The Book Paradigm for Improved Maintenance. IEEE Software, v.7, n.1, p.39-45, 1990

[OSB90] OSBORNE, W.M.; CHIKOFSKY, E.J. Fitting Pieces to the Maintenance Puzzle. IEEE Software, v.7, n.1, p.10-1, 1990

[OUR89] OURSTON, D. Program Recognition. IEEE Expert, v.4, n.4, p.36-49, 1989

[OXF86] Dictionary of Computing Oxford University Press, 1986

[PRE92] PRESSMAN, R.S. Software Engineering: A Practitioner's Approach. 3.ed. McGraw-Hill, 1992

[PEN95] PENTEADO, R. D. Uso, Evolução e Engenharia Reversa de um Ambiente de Apoio ao Desenvolvimento de Sistemas Reativos. Tese de Doutorado, IFSC-USP, 1995

[RAJ90] RAJLICH, V. Vifor Transforms Code Skeletons to Graphs. IEEE Software, v.7, n.3, p.60, 1990

[REK85] REKOFF, M.G. On Reverse Engineering. IEEE Transactions on Systems, Man, and Cybernetics, v.15, n.2, 1985

[RIC90] RICH, C.; WILLS, L.M. Recognizing a Program´s Design: A Graph-Parsing Approach. IEEE Software, v.7, n.1, p.82-9, 1990

[ROS88] ROSS, R.G. Entity Modeling: Techniques and Application. Data Research Group, 1988

[ROZ89] ROZENBLAT, G.D.; FISCHER, H. Reverse Engineering Technologies for Ada. In CASE '89: Proc. of the Third International Workshop on Computer-Aided Software Engineering, British Computer Society, p.560-74, 1989

[RUB91] RUMBAUGH, J. et al. Object-Oriented Modeling and Design. Prentice Hall International. Englewood Cliffs. 1991

[RUG90] RUGABER, S. et al. Recognizing Design Decisions in Programs. IEEE Software, v.7, n.1, p.46-54, 1990

[SAG95] SAGE, A.P. Systems Engineering and Systems Management for Reengineering. Journal Systems and Software, v.30, n.1, p.03-25, 1995

[SAM90] SAMUELSON, P. Reverse Engineering Someone Else´s Software: Is It Legal? IEEE Software, v.7, n.1, p.90-6, 1990

[SAN93] SANCHES, R., A Influência do Software e de seu Processo de Manutenção no Esforço de Manutenção. Tese de Doutorado, Universidade de São Paulo, Faculdade de Economia, Administração e Contabilidade, São Paulo, 1993

[SCA94] SCANDURA, J.M. Converting Legacy Code in Ada: A Cognitive Approach. IEEE Computer, v.27, n.4, p.55-61, 1994

[SCH87] SCHNEIDEWIND, N.F., The State of Software Maintenance. IEEE Transaction on Software Engineering, v.13, n.3, p. 303-10, 1987

[SCH94] SCHACH, S.R. The Economic Impact of Reuse on Maintenance. Journal Software Maintenance : Research and Practice. v.6, n.4, p.185-96, 1994

[SEV87] SEVIORA, R. Knowledge-Based Program Debugging Systems. IEEE Software, v.4, n.3, p.20-32, 1987

[SPI88] SPIVEY, J.M. Understanding Z: A Specification Language and its Formal Semantics. Cambridge University Press, 1988

[STA94] STAPLETON, L. Reverse Engineering Puzzles Out The Past. Open Computing, v.11, n.3, p.77-82, 1994

[STE95] STEPHEN, R.M.; LYNN, M.M. Software Migration and Reengineering: A Pilot Project in Reengineering. Journal Systems and Software, v.30, n.1, p.137-50, 1995

[SWA90] SWANSON, E.B.; BEATH, C.M., Departamentalization in Software Development and Maintenance. Communications of the ACM. v.33, n.6, p. 658-67, 1990

[VAL87] VALLBHANENI, S.R. Auditing the Maintenance of Software, Prentice-Hall, Inc., 1987.

[VAN90] VANEK, L.; DAVIS, L. Expert DataFlow and Static Analysis Tool. IEEE Software, v.7, n.3, p.63, 1990

[VIS94] VISAGGIO, G., Process Improvement Throught Data Reuse. IEEE Software, v.11, n.4, 1994.

[WAT94] WATERS, R.C.; CHIKOFSKY, E.J. Reverse Engineering : Progress Along Many Dimensions. Communications of the ACM. v.37, n.5, p.23-4, 1994.

[WIL90] WILDE, N., Dependency Analysis Tool Set Prototype. IEEE Software, v.7, n.3, p.65, 1990

[WIR90] WIRFS-BROCK; WILKERSON, V; WIENER, L. Designing Objec-Oriented Software. Prentice Hall. Englewood Cliffs. NJ. 1990

[WON95] WONG, K. et al. Structural Redocumentation: A Case Study. IEEE Software, v.12, n.1, p.46-54, 1995

[YAU85] YAU, S.S.; CHANG, P.S. Design Stability Measures for Software Maintenance. IEEE Transaction on Software Engineering, v.11, n.9, p.849-56, 1985

[YOU91] YOU, D. ACM SIGSOFT Software Engineering Notes. v.16, n.3, 1991