7 Estimativas de Esforço

Este capítulo vai apresentar algumas técnicas importantes e largamente usadas para estimação de esforço de desenvolvimento de software, porque não se pode planejar um projeto sem saber quanto tempo vai levar e quanto vai custar. Inicialmente é apresentada a técnica mais antiga, baseada em linhas de código ou *SLOC* (Seção 7.1). Em seguida são apresentados os conceitos fundamentais de uma técnica muito simples também baseada em linhas de código, o *COCOMO* (Seção 7.2). Depois, sua sucessora mais avançada *COCOMO II*, ou *CII*, adequada ao uso com o Processo Unificado, é apresentada em maior detalhe (Seção 7.3). Posteriormente são definidas as técnicas que não se baseiam em linhas de código, mas em funcionalidade aparente: *Análise de Pontos de Função* (Seção 7.4), a qual é bastante usada na indústria, *Pontos de Caso de Uso* (Seção 7.5), que vem se firmando por sua compatibilidade com o Processo Unificado e *Pontos de História* (Seção 7.6), a técnica preferida dos adeptos dos métodos ágeis.

Uma das questões fundamentais em um projeto de software é saber, antes de executá-lo, quanto esforço, em horas de trabalho, será necessário para levá-lo a termo. Essa área, chamada de *estimativa de esforço* conta com algumas técnicas que têm apresentado resultados interessantes ao longo dos últimos anos.

A maioria das técnicas de estimação de esforço utilizam pelo menos um parâmetro como base, por isso são chamadas de *técnicas paramétricas*.

Um exemplo de técnica *não paramétrica* é estimar que qualquer projeto de desenvolvimento sobre o qual não se sabe quase nada, vai levar *seis meses* para ser executado. Com o passar do tempo, à medida que mais informações sobre o projeto forem sendo disponibilizadas, esse tempo é ajustado para cima ou para baixo, baseando-se na opinião de especialistas. Apesar de essa técnica possivelmente ser bastante usada, poucos engenheiros de software conseguem boas previsões com ela.

Então, existem técnicas *paramétricas* baseadas em uma predição do número de linhas que o programa deverá ter e existem técnicas baseadas em requisitos, sejam eles descritos como funções, casos de uso ou histórias de usuário.

As técnicas de pontos de função, caso de uso e histórias baseiam-se em um conjunto de requisitos, aos quais é atribuído um peso que determinará a partir de certas transformações matemáticas o esforço necessário para seu desenvolvimento.

Já as técnicas baseadas em linhas de código necessitam de uma estimativa de quantas linhas de código deverão ser produzidas.

Existem também mecanismos que permitem converter parâmetros como pontos de função em estimativas de linhas de código, fazendo com que as técnicas acabem sendo compatíveis entre si.

Neste capítulo, inicialmente serão vistas as técnicas baseadas em linhas de código. Depois, pontos de função, pontos de caso de uso e pontos de histórias.

7.1 SLOC e KSLOC

A técnica conhecida como *LOC* (*Lines of Code*) ou *SLOC* (*Source Lines of Code*) foi possivelmente a primeira a surgir e consiste em estimar o número de linhas que um programa deverá ter, normalmente a partir da opinião de especialistas e histórico de projetos passados.

Este tipo de técnica surgiu numa época em que as linguagens de programação, como FORTRAN eram fortemente orientadas a linhas de código. Naquele tempo, programas eram registrados em cartões perfurados, uma linha de código por cartão, de forma que a altura da pilha de cartões era uma medida bastante natural para a complexidade de um programa.

Rapidamente a técnica evoluiu para a forma conhecida como *KSLOC* (*Kilo Source Lines of Code*), tendo em vista que o tamanho da maioria dos programas passou a ser medido em milhares de linhas. Uma unidade KSLOC, portanto, vale mil unidades SLOC. Além disso, também são usados os termos MSLOC para milhões de linhas e GSLOC para bilhões de linhas de código.

Porém, as linguagens atuais não são mais tão restritivas em termos de linhas de código, de forma que talvez fizesse mais sentido falar em comandos ao invés de linhas. Usuários de linguagens como C, Java ou Pascal, por exemplo, poderão contar o número de comandos terminados por ";", obtendo assim uma boa medida de complexidade de um programa. Mesmo assim, a noção de comando só funciona bem para linguagens imperativas. No caso de linguagens declarativas ou funcionais outras formas de medir complexidade precisam ser usadas.

Mesmo assim, a noção de linha de código continua sendo uma medida popular para complexidade de programas, que podem variar de 10 a 100.000.000 de linhas, com complexidade inerente diretamente proporcional na maioria das vezes.

De fato, a medida faz sentido quando se compara ordens de magnitude. Um programa com 100.000 linhas possivelmente será mais complexo do que um programa com 10.000 linhas. Mas pouco se pode concluir ao se comparar um programa com 10.000 linhas e um programa com 11.000 linhas.

7.1.1 Estimação de KSLOC

Uma técnica para estimação de KSLOC é reunir a equipe para discutir o sistema a ser desenvolvido. Cada participante dará a sua opinião sobre a quantidade de KSLOC que serão necessárias para desenvolver o sistema. Usualmente a reunião não chegará a um valor único. Deverão então ser considerados pelo menos 3 valores:

a) O KSLOC *otimista*, ou seja, o número mínimo de linhas que se espera desenvolver se todas as condições forem favoráveis.
b) O KSLOC *pessimista*, ou seja, o número máximo de linhas que se espera desenvolver ante condições desfavoráveis.
c) O KSLOC esperado, ou seja, o número de linhas que efetivamente se espera desenvolver em uma situação de normalidade.

A partir desses valores, o KSLOC pode ser calculado assim:

$$KSLOC = (4*KSLOC_{esperado} + KSLOC_{otimista} + KSLOC_{pessimista}) / 6$$

Essas estimativas devem ser, depois comparadas com a informação real, ao final do projeto, e desta forma a equipe deve ter um *feedback* para que possa ajustar futuras previsões. Por exemplo, um membro que usualmente prevê valores muito abaixo do real, deverá perceber que está sendo muito otimista e tentar ajustar suas previsões para valores mais altos no futuro.

Uma técnica que pode ser empregada para ajustar a capacidade de previsão da equipe é tomar projetos já desenvolvidos, para os quais se saiba de antemão a quantidade e linhas, e exercitar a estimação com a equipe para estes projetos, comparando depois os valores reais com os valores estimados por cada um.

Erros de previsão na segunda ou terceira casa (20 ou 30%) não são significativos, ou seja, é perfeitamente aceitável fazer uma previsão de 10 mil linhas para um projeto de 12 ou 13 mil linhas. Mas seria um problema caso a equipe fizesse uma previsão de 10 mil linhas para um projeto que ao final tivesse 30 ou 40 mil linhas.

A medida de número de linhas, porém ainda pode ser traiçoeira. Programadores experientes produzirão software com possivelmente menos linhas do que programadores menos experientes. A reutilização de software baixa o número de linhas efetivamente produzidas em relação às funcionalidades disponibilizadas. KSLOC usualmente não deve contar linhas de código reusadas, mas apenas aquelas que são efetivamente produzidas como código novo, ou que, se reusadas, são modificadas.

Alguns cuidados devem ser tomados. Não se deve considerar SLOC como uma boa medida de produtividade individual, pois muitas vezes a complexidade inerente de um programa vai muito além da quantidade de linhas: poucas linhas de código poderão realizar atividades difíceis e complexas enquanto muitas linhas poderão efetuar apenas atividades triviais. É o caso de quem desenvolve software científico ou jogos em contraponto com a produção de meros cadastros e relatórios.

Outra situação a considerar é o fato de que refatorações do software poderão remover muitas linhas de código inútil ou redundante e isso não deve ser considerado como uma produtividade negativa.

7.1.2 Transformando Pontos de Função em KSLOC

Existem estudos (Jones C. , 1996) que apresentam uma relação entre o número de pontos de função não ajustados (UFP – Seção 7.3.3) e o número de linhas de código que se pode esperar em média conforme a linguagem de programação (*backfire table*), conforme mostrado na Tabela 7-1.

Tabela 7-1: *Backfire table* para conversão de UFP em SLOC[84].

Linguagem	Default SLOC/UFP	Linguagem	Default SLOC/UFP
Access	38	Jovial	107
Ada 83	71	Lisp	64
Ada 95	49	Código de Máquina	640
AI Shell	49	Modula 2	80
APL	32	Pascal	91
Assembly – básico	320	Perl	27
Assembly – macro	213	PowerBuilder	16
Basic – ANSI	64	Prolog	64
Basic compilado	91	Query	13
C	128	Gerador de relatórios	80
C++	55	Linguagem de simulação	46
Cobol	91	Planilha	6
Forth	64	Scripts da Shell do Unix	107
HTML 3.0	15	Visual Basic	29
Java	53	Visual C++	34

Assim, por exemplo, um sistema que, pela técnica de Análise de Pontos de Função tenha seu tamanho estimado em 1000 pontos de função terá um tamanho em SLOC estimado em 53 mil linhas Java, ou 91 mil linhas Pascal, ou ainda 55 mil linhas C++.

Estes valores são valores médios e foram obtidos a partir da análise estatística de vários projetos feita por Jones.

7.1.3 Como Contar Linhas de Código

Dependendo da linguagem de programação, poder-se-ia perguntar o que efetivamente conta como linha de comando. A ideia é que apenas linhas lógicas contem. Mas ainda assim ficariam algumas dúvidas: “else” conta? Declaração de variável conta? Park (1992) apresenta para várias linguagens de programação, quais tipos de comandos devem efetivamente contados e quais não devem ser contados como SLOC. Uma versão atualizada e revisada destas orientações para uso com COCOMO II (Seção 7.3) podem ser encontradas em forma de tabela no trabalho de Boehm (2000)[85], as qual é resumida abaixo.

Em relação ao tipo de comando devem ser contados:

a) Comandos executáveis.
b) Declarações.
c) Diretivas de compilação.

Por outro lado, não devem ser contados:

a) Comentários.
b) Linhas em branco.

Em relação à forma como o código é produzido, devem ser contadas as linhas:

[84] Fonte: Jones (1996).
[85] csse.usc.edu/csse/research/COCOMOII/cocomo2000.0/CII_modelman2000.0.pdf

a) Programadas.
b) Copiadas ou reusadas.
c) Modificadas.

Não devem ser contadas linhas:

a) Geradas por geradores automáticos de código.
b) Removidas.

Em relação aos comandos presentes na maioria das linguagens de programação, devem ser contados:

a) Comandos *null*, *continue* e *no-op*.
b) Comandos que instanciam elements genéricos.
c) Pares *begin-end* ou {...} usados em comandos estruturados.
d) Comandos *elseif*.
e) Palavras chave como *division*, *interface* e *implementation.*

Não devem ser contados:

a) Comandos vazios como "; ", quando colocados sozinhos em uma linha.
b) Pares *begin-end* ou {...} usados em para delimitar o bloco principal ou procedimentos e funções.
c) Expressões passadas como argumentos para chamadas de procedimentos ou função (conta-se apenas a chamada).
d) Expressões lógicas em comandos IF, WHILE ou REPEAT (conta-se apenas o comando que contém as expressões).
e) Símbolos que servem como finalizadores de comandos executáveis, declarações ou subprogramas.
f) Símbolos THEN, ELSE e OTHERWISE, quando aparecerem sozinhos em uma linha (mas conta-se o comando que os sucede).

São entendidos como *comandos executáveis*, e devem ser contados, todos os comandos que sejam atribuições, *GOTO*, chamada de procedimento, chamada de macro, retorno, *break*, *exit*, *stop*, *continue*, *null*, *noop*, etc. Também devem ser contadas separadamente as estruturas de controle como estruturas de repetição e seleção, e inclusive seus blocos *begin-end* ou {...}, se exisitirem, contam separadamente. Por exemplo, o fragmento de código abaixo deve contar como 4 linhas de código:

```
BEGIN                              0
  IF (a <> b) OR (b < c) THEN     +1
    BEGIN                         +1
      X := b;                     +1
    END                            0
  ELSE                             0
    X := fatorial(a)              +1
  ;                                0
END;                               0
                                   ---
                                   4
```


As declarações são comandos não executáveis que também devem ser contados. Por exemplo, declarações de nomes, números, constantes, objetos, tipos, subtipos, programas, subprogramas, tarefas, exceções, pacotes, genéricos e macros. Blocos *begin-end* ou {...} quando são parte obrigatória da declaração de subprogramas, não devem ser contados separadamente (apenas a declaração do subprograma conta). Assim, um bloco como o seguinte deve contar como 15 linhas:

```
procedure fib(num:integer):integer;                +1
var ultimoFib, cont, penultimoFib : integer;       +1
begin                                               0
   if num = 0 then                                 +1
     fib := 0                                      +1
   else                                             0
     if num = 1 then                               +1
       fib := 1                                    +1
     else                                           0
       Begin                                       +1
         ultimoFib := 1;                           +1
         cont := 1;                                +1
         repeat                                    +1
           penultimoFib := ultimoFib;              +1
           ultimoFib := fib;                       +1
           fib := penultimoFib + ultimoFib;        +1
           cont := cont + 1;                       +1
         until cont = num;                         +1
       end                                          0
     ;                                              0
   ;                                                0
end;                                                0
                                                   ----
                                                   15
```

Nota-se que apesar de três variáveis terem sido declaradas, elas o foram em uma única linha, contando, portanto, uma única vez. Se fossem dois tipos de variáveis (por exemplo, inteiras e reais), seriam necessárias duas declarações, e contaria uma linha para cada uma:

```
VAR x, y : integer;   +1
    Z : real;         +1
                      ---
                       2
```

7.2 COCOMO

COCOMO ou *Constructive Cost Model* (também conhecido como COCOMO 81) é um modelo de estimativa de esforço baseado em KSLOC. Este modelo já é obsoleto, e foi substituído por COCOMO II em aplicações reais, mas ainda assim é apresentado aqui porque sua simplicidade conceitual ajuda a esclarecer conceitos usados por outras técnicas, e ainda pode ser usado como uma ferramenta de estimação grosseira caso pouquíssima informação sobre o sistema esteja disponível.

O modelo COCOMO foi criado por Boehm (1981) a partir de um estudo empírico sobre sessenta e três projetos na empresa TRW Aerospace. Os programas examinados tinham de 2 a 100 KSLOC e eram escritos em linguagens tão diversas quanto Assembly e PL/I.

O modelo COCOMO 81 apresenta-se em três implementações de complexidade crescente de acordo com o grau de informações que se tenha a respeito do sistema a ser desenvolvido:

a) *Implementação básica*, quando a única informação sobre o sistema efetivamente disponível é o número estimado de linhas de código.
b) *Implementação intermediária*, quando certos fatores relativos ao produto, suporte computacional, pessoal e processo são conhecidos e podem ser avaliados para o sistema a ser produzido.
c) *Implementação avançada*, quando for necessário subdividir o sistema em subsistemas e distribuir as estimativas de esforço por fase e atividade.

Para o cálculo do esforço, todas as implementações do COCOMO consideram também o tipo de projeto a ser desenvolvido:

a) *Modo orgânico*, que se aplica quando o sistema a ser desenvolvido não envolver dispositivos de hardware e a equipe estiver acostumada a desenvolver este tipo de aplicação, ou seja, sistemas de baixo risco tecnológico e baixo risco de pessoal.
b) *Modo semidestacado*, que se aplica a sistemas com maior grau de novidade para a equipe e que envolvem interações significativas com hardware, mas para os quais a equipe ainda tem algum conhecimento, ou seja, sistemas onde a combinação do risco tecnológico e de pessoal seja médio.
c) *Modo embutido*, que se aplica a sistemas com alto grau de interação com diferentes dispositivos de hardware, ou que sejam embarcados, e para os quais a equipe tenha considerável dificuldade de abordagem. São os sistemas com alto risco tecnológico e/ou de pessoal.

A Tabela 7-2 apresenta uma sugestão para determinar o tipo de projeto a partir dos riscos gerais identificados. Nos riscos de pessoal incluem-se também aspectos referentes ao cliente e aos requisitos. Requisitos altamente voláteis indicam um projeto de alto risco.

Tabela 7-2: Combinação de risco de pessoal e risco tecnológico para a escolha do tipo de projeto no contexto de COCOMO.

Risco tecnológico **Risco de pessoal**	**Alto**	**Médio**	**Baixo**
Alto	Embutido	Embutido	Semidestacado
Médio	Embutido	Semidestacado	Orgânico
Baixo	Semidestacado	Orgânico	Orgânico

As três implementações do modelo COCOMO permitem determinar 3 informações básicas:

a) O esforço estimado em desenvolvedor-mês: *E*.
b) O tempo linear de desenvolvimento sugerido em meses corridos: *D*.
c) O número médio de pessoas recomendado para a equipe: *P*.

O número de pessoas para a equipe será sempre dado por $P = E/D$. A medida parece simplista, porque aparentemente considera que a relação entre o número de pessoas e o tempo de projeto é linear. Mas o cálculo do esforço estimado (E) já leva em conta a não linearidade dessa relação, pois é uma função exponencial, como será visto adiante.

Além disso, é importante que nas fórmulas de COCOMO e CII seja usada sempre a unidade *desenvolvedor-mês*. Variações como *desenvolvedor-semana*, *desenvolvedor-dia* ou *desenvolvedor-hora* poderão provocar distorções nos resultados devido ao uso de exponenciais nas fórmulas.

7.2.1 Modelo Básico

A implementação mais simples de COCOMO é capaz de calcular esforço, tempo e tamanho de equipe a partir de uma simples estimativa de KSLOC.

Inicialmente calcula-se o esforço (E) a partir da seguinte fórmula:

$$E = ab * KSLOC^{bb}$$

onde, *ab* e *bb* são obtidos a partir da tabela mostrada na onde *cb* e *db* são constantes também dadas na **Erro! Auto-referência de indicador não válida.**.

Tabela 7-3.

Já o tempo linear ideal recomendado para o desenvolvimento é dado por:

$$T = cb * E^{db}$$

onde *cb* e *db* são constantes também dadas na **Erro! Auto-referência de indicador não válida.**.

Tabela 7-3: Valores de *ab*, *bb*, *cd* e *db* em função do tipo de projeto.

Tipo de projeto	*ab*	*bb*	*cb*	*db*
Orgânico	2,4	1,05	2,5	0,38
semidestacado	3,0	1,12	2,5	0,35
Embutido	3,6	1,2	2,5	0,32

Por exemplo, um projeto com KSLOC estimado em 20 (20.000 linhas de código) com baixo risco (modo orgânico), produzirá as seguintes estimativas:

$E = 2{,}4 * 20^{1{,}05} = 56$ desenvolvedor-mês

$T = 11{,}5$ meses

$P = 5$ pessoas

Todos os valores são aproximados devido às casas decimais, mas a conclusão do modelo COCOMO básico é que um projeto orgânico com previsão de 20.000 linhas de código será desenvolvido em cerca de 1 ano por uma equipe de cerca de 5 pessoas.

O modelo básico é bom por ser simples e rápido, mas sua capacidade preditiva é limitada devido ao fato de que o esforço de desenvolvimento não ser função apenas do número de

linhas de código, mas também de outros fatores, que são considerados a partir do modelo intermediário.

7.2.2 Modelo Intermediário

O modelo intermediário de COCOMO vai considerar uma avaliação sobre vários aspectos do projeto além das simples linhas de código. A tabela mostrada na Tabela 7-4 apresenta os 15 fatores do modelo original organizados em 4 grupos. A tabela exige que se indique para cada fator uma nota que varia de "muito baixa" a "extra-alta". A partir dessa nota, um valor numérico correspondente é obtido para cada fator. Por exemplo, se a dimensão da base de dados esperada é "muito alta" então o fator DATA terá valor numérico 1,16.

Nota-se que nem todas as notas são aplicáveis a todos os fatores. Por exemplo, o fator DATA não pode ter notas "muito baixa" nem "extra-alta".

Tabela 7-4: Fatores influenciadores de custo do modelo COCOMO intermediário.

Fatores influenciadores de custo	Acrônimo	Muito baixa	Baixa	Média	Alta	Muito Alta	Extra Alta
Relativos ao produto							
Nível de confiabilidade requerida	RELY	0,75	0,88	1,00	1,15	1,40	
Dimensão da base de dados	DATA		0,94	1,00	1,08	1,16	
Complexidade do produto	CPLX	0,70	0,85	1,00	1,15	1,30	1,65
Suporte computacional							
Restrições ao tempo de execução	TIME			1,00	1,11	1,30	1,66
Restrições ao espaço de armazenamento	STOR			1,00	1,06	1,21	1,56
Volatilidade da máquina virtual	VIRT		0,87	1,00	1,15	1,30	
Tempo de resposta do computador	TURN		0,87	1,00	1,07	1,15	
Pessoal							
Capacidade dos analistas	ACAP	1,46	1,19	1,00	0,86	0,71	
Experiência no domínio da aplicação	AEXP	1,29	1,13	1,00	0,91	0,82	
Capacidade dos programadores	PCAP	1,42	1,17	1,00	0,86	0,70	
Experiência na utilização da máquina virtual	VEXP	1,21	1,10	1,00	0,90		
Experiência na linguagem de programação	LEXP	1,14	1,07	1,00	0,95		
Processo							
Adoção de boas práticas de programação	MODP	1,24	1,10	1,00	0,91	0,82	
Uso de ferramentas atualizadas	TOOL	1,24	1,10	1,00	0,91	0,83	
Histórico de projetos terminados no prazo	SCED	1,23	1,08	1,00	1,04	1,10	

Os 15 valores numéricos obtidos para os fatores são combinados através de multiplicação, produzindo o valor *EAF*:

$$EAF = RELY * DATA * CPLX * TIME * ... * SCED$$

O modelo intermediário então combina essa estimativa com *KSLOC* da seguinte forma:

$$E = ai * KSLOC^{bi} * EAF$$

onde os índices *ai* e *bi* são dados de acordo com a Tabela 7-5.

Tabela 7-5: Valores de *ai* e *bi* em função do tipo de projeto.

Tipo de projeto	*ai*	*bi*
orgânico	2,8	1,05
semidestacado	3,0	1,12
embutido	3,2	1,2

No modelo COCOMO intermediário a avaliação dos fatores influenciadores do custo é dada de forma intuitiva, o que pode levar a certa subjetividade da avaliação. O modelo COCOMO II é bem mais detalhado em relação à forma como essas notas são atribuídas, mas também é bem mais trabalhosa a sua aplicação. Uma sugestão a ser seguida na atribuição dessas notas é que somente seja atribuída uma nota extrema caso não se consiga imaginar uma situação ainda mais extrema. Por exemplo, qual o nível de confiabilidade requerida para um sistema de videolocadora? Muito alta? Neste caso, que avaliação deveria ser dada ao sistema embarcado em uma aeronave, ou um sistema de tele-cirurgia? Por outro lado, a atribuição do conceito muito baixo para RELY no caso da videolocadora não significa que será gerado um sistema não confiável, significa apenas que não serão tomadas atitudes especiais de garantia de confiabilidade que seriam tomadas no caso de sistemas que colocam vidas em risco.

Além disso, é sempre possível fazer duas avaliações para os fatores influenciadores de custo, quando não se tem certeza sobre a melhor nota a ser atribuída. Por exemplo, na versão otimista, a nota de RELY poderia ser "baixa" e na pessimista "média". Deve-se observar, porém, que a atribuição de notas diferentes para muitos fatores poderá levar a resultados bem diferentes entre si, o que aumentaria bastante o fator de incerteza quanto à duração do projeto.

Por exemplo, no cenário otimista da Tabela 7-6 o valor de EAF será 1,3267. Já no cenário pessimista o EAF será 2,8497.

Tabela 7-6: Aplicação de notas aos fatores influenciadores de custo em cenário otimista e pessimista.

Fator	Otimista	Nota	Pessimista	Nota
RELY	Média	1,00	Alta	1,15
DATA	Muito Alta	1,16	Muito Alta	1,16
CPLX	Média	1,00	Média	1,00
TIME	Alta	1,11	Muito Alta	1,30
STOR	Alta	1,06	Muito Alta	1,21
VIRT	Baixa	0,87	Baixa	0,87
ACAP	Média	1,00	Média	1,00
AEXP	Média	1,00	Média	1,00
PCAP	Média	1,00	Média	1,00
VEXP	Alta	0,90	Média	1,00
LEXP	Alta	0,95	Média	1,00
MODP	Baixa	1,10	Baixa	1,10
TOOL	Baixa	1,10	Muito Baixa	1,24
SCED	Baixa	1,08	Muito Baixa	1,23
Produto		**1,33**		**3,06**

Nota-se que a interpretação de otimismo ou pessimismo depende do fator, nos primeiros sete fatores, quanto mais alta a nota, mais pessimismo, nos oito fatores seguintes, quanto mais alta a nota, mais otimismo. Em caso de dúvida, deve-se ter em mente que o valor numérico da opção pessimista sempre deve ser igual ou maior que o valor numérico da opção otimista.

Assim, no cenário otimista o esforço previsto para um projeto orgânico com 20 *KSLOC* será:

$$E = 2{,}8 * 20^{1{,}05} * 1{,}33 = 86 \text{ desenvolvedor-mês}$$

Já no cenário pessimista o esforço seria:

$$E = 2{,}8 * 20^{1{,}05} * 3{,}06 = 199 \text{ desenvolvedor-mês}$$

Essa discrepância mostra como esses fatores podem afetar a produtividade da equipe em função do projeto.

7.2.3 Modelo Avançado

A implementação avançada ou completa do modelo COCOMO introduz facetas como a decomposição do projeto em subprojetos, bem como estimativas individualizadas para as fases do projeto.

Porém, como o modelo é complexo e desatualizado, sua apresentação será omitida, em função da apresentação de seu sucessor COCOMO II na seção seguinte.

7.3 COCOMO II

COCOMO II[86], ou *CII* é uma evolução do antigo modelo COCOMO 81 e ao contrário de seu antecessor, funciona bem com ciclos de vida iterativos e é fortemente adaptado para uso com o Processo Unificado (Boehm B. , 2000), embora também seja definido para os modelos Cascata e Espiral.

O CII foi projetado para mensurar o esforço e o tamanho médio de equipe para as fases de elaboração e construção do Processo Unificado. O esforço e equipe para as fases de concepção e transição podem ser então calculados como uma fração dos valores obtidos para as duas outras fases. A Figura 7-1 apresenta esquematicamente a região de estimação de esforço abrangida pelo método CII.

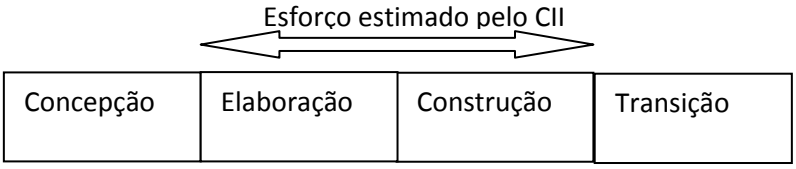

Figura 7-1: A Região de estimação de esforço de COCOMO II.

Como se pode ver na figura, CII é aplicado para determinar o esforço necessário para desenvolver as fases de Elaboração e Construção de um projeto. A duração das fases de Concepção (caso ainda não tenha terminado) e Transição deve ser feita pela aplicação de um percentual do esforço calculado para das duas outras fases (Seção 7.3.3).

O método CII define o esforço total, em desenvolvedor-mês, inicialmente a partir do número do KSLOC, de uma constante ajustável por dados históricos *A*, de um valor que pode ser calculado para cada projeto, chamado de expoente de esforço *S* e por um conjunto de fatores multiplicadores de esforço M_i, os quais também são individualmente calculados para cada projeto a partir de notas dadas. A equação geral que define o esforço total nominal de um projeto é a seguinte:

[86] sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html

$$E = A * KSLOC^S * \prod_{i=1}^{n} M_i$$

Onde:

a) *E* é o esforço total nominal que se deseja calcular para o projeto (fases de elaboração e construção).
b) *A* é uma constante que deve ser calibrada a partir de dados históricos. CII sugere um valor inicial de 2,94.
c) *KSLOC* é o número estimado de milhares de linhas de código que deverão ser desenvolvidas.
d) *S* é o coeficiente de esforço, cujo cálculo é mostrado abaixo.
e) M_i são os multiplicadores de esforço, que são explicados na Seção 7.3.2.

O expoente de esforço *S* é calculado a partir de uma constante *B*, que deve ser ajustada a partir de dados históricos e de um conjunto de cinco fatores de escala F_j. O cálculo se dá da seguinte forma:

$$S = B + 0{,}001 * \sum_{j=1}^{5} F_j$$

Onde:

a) *S* é o expoente de esforço que se deseja calcular.
b) B é uma constante que deve ser calibrada de acordo com valores históricos. CII sugere um valor inicial de 0,91.
c) F_j são cinco fatores de escala (*scale factors*) que devem ser atribuídos para cada projeto específico, tomando-se sempre muito cuidado pois sua influência no cálculo do esforço total do projeto é exponencial.

Ainda em relação aos fatores multiplicadores de esforço M_i, sua quantidade varia em função de se estar calculando o esforço nas fases iniciais ou intermediárias do projeto.

Durante as fases de Concepção e início da Elaboração usa-se o *Early Design Model* (Seção 0), com 6 fatores (*n*=6). Mais tarde pode-se usar o *Post-Architecture Model* (Seção 7.3.2.1), com 16 fatores (*n*=16). A Figura 7-2 resume o momento em que cada um dos modelos deve ser usado, sendo que na fase de Elaboração, a decisão por um modelo ou outro vai depender de quanto já se tenha avançado nesta fase.

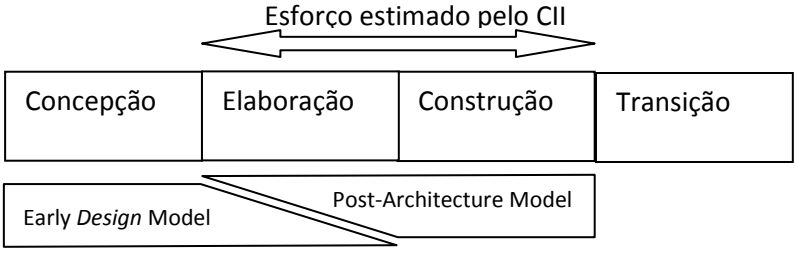

Figura 7-2: Momento de aplicação do *early design* e do *post-architecture models*.

Nota-se que, se qualquer um dos modelos for aplicado durante a fase de elaboração ou construção o modelo vai prever o esforço total. Então o esforço já despendido deverá ser descontado para que a previsão diga respeito ao tempo ainda restante do projeto.

O valor E corresponde, então, ao esforço total em desenvolvedor-mês para estas fases do UP. Mas, normalmente, projetos são desenvolvidos por equipes e não por uma única pessoa. Assim, uma equipe maior pode conseguir desenvolver um projeto mais rápido do que uma equipe menor. Mas existe um limite a partir do qual a equipe será grande demais para o tamanho do projeto e o esforço de gerenciar a equipe não compensará mais o ganho com tempo, havendo inclusive, uma inversão da curva no sentido de que quanto maior a equipe, maior será o tempo do projeto (Figura 7-3).

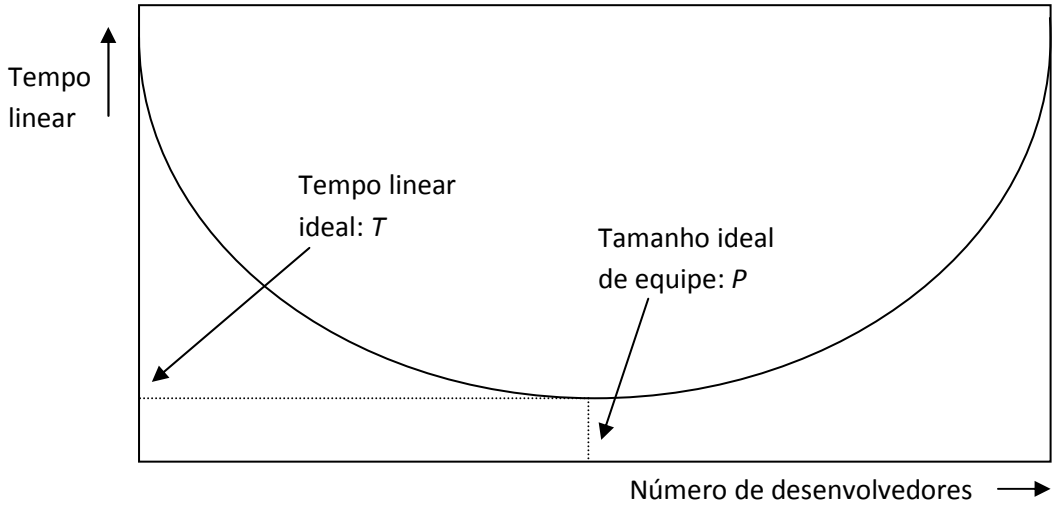

Figura 7-3: Relação entre o tamanho da equipe e o tempo linear de desenvolvimento de um projeto.

Pode-se falar, assim, em um tempo ideal e um tamanho de equipe ideal para desenvolver um projeto. O CII sugere que o tempo linear ideal para desenvolver um projeto cujo esforço total *E* já é conhecido seja calculado a partir da seguinte fórmula:

$$T = C * (E)^{D+0,2*(S-B)}$$

Onde:

a) *T* é o tempo linear ideal de desenvolvimento.
b) *B, C* e *D* são constantes que devem ser calibradas a partir de dados históricos (ver abaixo).
c) *E* é o esforço total para o projeto, conforme calculado anteriormente.
d) *S* é o expoente de esforço que já foi mencionado.

Um desenvolvedor-mês equivale a aproximadamente 152 horas de trabalho, uma conta que já exclui valores médios de fins de semana, feriados e férias.

Finalmente, o *tamanho médio da equipe P* é obtido pela simples divisão do esforço total pelo tempo linear:

$$P = E/T$$

Essa fórmula, porém, só se aplica quando o tempo *T* for efetivamente o tempo linear ideal. De acordo com a Figura 7-3, pode-se perceber que uma redução no tempo linear não implica em um aumento de equipe proporcional, ou seja, para fazer o projeto na metade do tempo, não adianta dobrar o tamanho da equipe.

O método COCOMO II possui um multiplicador de esforço específicamente para indicar essa relação. O multiplicador SCED (cronograma de desenvolvimento requerido), apresentado na Seção 7.3.2.1, tem valor nominal igual a 1,0 se o projeto deve ser desenvolvido no seu tempo ideal. Caso se queira forçar um desenvolvimento mais rápido, em 85% do tempo ideal, SCED vai valer 1,14. Caso se queira forçar um desenvolvimento ainda mais rápido, em 75% do tempo ideal, SCED passa a valer 1,43, ou seja, indicando um esforço 43% maior para obter uma redução de menor do que 25% no tempo linear. Isso porque o novo tempo linear será maior do que 25% do tempo originalmente calculado já que o esforço total também será maior.

Os valores das constantes *A*, *B*, *C* e *D* foram obtidos pela equipe da *USC* (*University of Southern California*) a partir da análise de 161 projetos. Tais valores são os seguintes:

a) *A* = 2,94
b) *B* = 0,91
c) *C* = 3,67
d) *D* = 0,28

Porém, recomenda-se que pelo menos os valores de *A* seja calibrados a partir de dados obtidos na organização local que estiver usando a técnica (Seção 0).

7.3.1 Fatores de Escala

Os cinco fatores de escala mencionados anteriormente receberão cada um uma nota que varia de “muito baixo” até “extremamente alto”. Os fatores de escala terão impacto exponencial no tempo de desenvolvimento. Se os fatores de escala forem nominais (= 1,0), então estima-se que um projeto com 200 KSLOC terá um esforço duas vezes maior do que um projeto com 100 KSLOC, ou seja, o crescimento do esforço é proporcional ao tamanho do projeto. Em outras palavras, mesmo que o projeto aumente de tamanho, o custo de cada linha de código permanece inalterado.

Se os fatores de escala ficarem acima do nominal (> 1,0), então um projeto com 200 KSLOC vai usar *mais do que o dobro* do esforço do que um projeto com 100 KSLOC, ou seja, quanto maior o projeto, mais cara se torna cada linha de código.

Por outro lado, se os fatores de escala ficarem abaixo do nominal (< 1,0), então um aumento do tamanho do projeto vai proporcionar um ganho em escala, ou seja, quanto maior o projeto, mais barata se torna cada linha de código.

Os fatores de escala são os seguintes:

a) *Precedentes* (*PREC*): Se o produto é similar a vários projetos desenvolvidos anteriormente, então PREC é alto.

b) *Flexibilidade no Desenvolvimento* (*FLEX*): Se o produto deve ser desenvolvido estritamente dentro dos requisitos, é preso a definições de interfaces externas, então FLEX é baixo.
c) *Arquitetura/Resolução de Riscos* (*RESL*): Se existe bom suporte para resolver riscos e para definir a arquitetura então RESL é alto.
d) *Coesão da Equipe* (*TEAM*): Se a equipe é bem formada e coesa, então TEAM é alto.
e) *Maturidade de Processo* (*PMAT*): Este fator pode estar diretamente associado com o nível de maturidade CMMI (Seção 12.3.3). Quanto mais alto o nível de maturidade, maior será PMAT.

As tabelas abaixo são sugestões de Boehm sobre como definir a nota de cada um dos fatores de escala considerados. Para cada fator, toma-se a primeira tabela e decide-se qual a avaliação para cada uma das características na primeira coluna. Após obter todas as avaliações (uma para cada linha), determina-se uma nota para o fator como um todo, escolhendo a coluna mais representativa (muito baixo, baixo, nominal, alto, muito alto ou extremamente alto). A escolha da coluna mais representativa não é feita de maneira formal, ou seja, a ponderação dos requisitos será feita pelo planejador de projeto de acordo com sua percepção e experiência. Mas a princípio, deve-se considerar a coluna que represente a média das notas dadas.

Uma vez selecionada a avaliação (coluna) para o fator de escala, usa-se a segunda tabela para obter sua equivalente numérica, a qual será usada na fórmula do cálculo do esforço *E*, conforme mostrado no início deste capítulo.

Para *PREC* (*o produto é similar a produtos anteriormente desenvolvidos pela mesma equipe*), a Tabela 7-7, em duas partes, apresenta os padrões para atribuição do equivalente numérico.

Tabela 7-7: Forma de obtenção do equivalente numérico para PREC.

Característica	**Muito baixo** **Baixo**	**Nominal** **Alto**	**Muito alto** **Extra-alto**
Compreensão organizacional dos objetivos do produto	Geral	Considerável	Total
Experiência no trabalho com sistemas de software relacionados	Moderada	Considerável	Extensiva
Desenvolvimento concorrente de novo hardware e procedimentos operacionais associados	Extensivo	Moderado	Algum
Necessidade de arquiteturas e algoritmos de processamento de dados inovadores	Considerável	Algum	Mínimo

Nota média	**Muito baixo**	**Baixo**	**Nominal**	**Alto**	**Muito alto**	**Extra-alto**
Interpretação	Totalmente sem precedentes	Largamente sem precedentes	Um tanto sem precedentes	Genericamente familiar	Altamente familiar	Totalmente familiar
Fator numérico	6,20	4,96	3,72	2,48	1,24	0,00

Assim, por exemplo, imaginando que um projeto tenha a seguinte avaliação:

a) Compreensão organizacional dos objetivos do produto: *considerável*.
b) Experiência no trabalho com sistemas de software relacionados: *moderada*.
c) Desenvolvimento concorrente de novo hardware e procedimentos operacionais associados: *moderado*.

d) Necessidade de arquiteturas e algoritmos de processamento de dados inovadores: *algum*.

Neste caso, tem-se três notas na coluna Nominal/Alto e uma nota na coluna Muito baixo/Baixo. Não faria sentido atribuir notas Muito baixo (porque apenas uma nota contra três está nesta coluna), nem Muito alto ou Extra-alto (porque nenhuma nota está nestas colunas). Como são três notas na coluna Nominal/Alto e uma na coluna Muito baixo/Baixo, pode-se atribuir nota "Nominal" a PREC (o limite inferior da segunda coluna). Assim, a interpretação de PREC vai considerar que o projeto é "um tanto sem precedentes" e o fator de escala numérico será 3,72.

A Tabela 7-8 apresenta a forma de cálculo do fator de escala *FLEX*, ou seja, qual a *flexibilidade no desenvolvimento* em relação aos requisitos.

Tabela 7-8: Forma de obtenção do equivalente numérico para FLEX.

Característica	**Muito baixo** **Baixo**	**Nominal** **Alto**	**Muito alto** **Extra-alto**
Necessidade de conformação do software a requisitos pré-estabelecidos	Total	Considerável	Básica
Necessidade de conformação do software a especificações de interfaces com sistemas externos	Total	Considerável	Básica
Combinação das inflexibilidades acima com prêmio por término antecipado do projeto	Alto	Médio	Baixo

Nota média	**Muito baixo**	**Baixo**	**Nominal**	**Alto**	**Muito alto**	**Extra-alto**
Interpretação	Rigoroso	Relaxamento ocasional	Algum relaxamento	Conformidade geral	Alguma conformidade	Metas gerais
Fator numérico	5,07	4,05	3,04	2,03	1,01	0,00

Interpretando-se as tabelas acima depreende-se que quanto maior o rigor em relação à conformidade com os requisitos, ou seja, quanto menor a flexibilidade do projeto, mais tempo ele vai levar para ser desenvolvido. Isso é natural, uma vez que as atividades de projeto incluem não apenas a escrita de código, mas todas as atividades, inclusive as inspeções de conformidade e testes exaustivos.

O fator *RESL*, ou seja, a existência de arquitetura ou sistema de suporte para *resolução de riscos* é calculado de acordo com os dados da Tabela 7-9.

Tabela 7-9: Forma de obtenção do equivalente numérico para RESL.

Característica	Muito baixo	Baixo	Nominal	Alto	Muito alto	Extra-alto
O plano de gerenciamento de risco identifica todos os itens de risco críticos e estabelece marcos para resolvê-los	Nada	Um pouco	Alguma coisa	Geralmente	Largamente	Totalmente
Cronograma, orçamento e marcos internos são compatíveis com o plano de gerenciamento de risco	Nada	Um pouco	Alguma coisa	Geralmente	Largamente	Totalmente
Percentual do cronograma de desenvolvimento devotado a estabelecer a arquitetura, uma vez definidos os objetivos gerais do produto	5	10	17	25	33	40
Percentual de arquitetos de software experientes (*top*) disponíveis para o projeto em relação ao considerado necessário	20	40	60	80	100	120
Suporte de ferramentas disponível para resolver itens de risco, desenvolver e verificar especificações arquiteturais	Nenhum	Pouco	Algum	Bom	Forte	Total
Nível de incerteza nos determinantes-chave da arquitetura: missão, interface com usuário, COTS, hardware, tecnologia, desempenho	Extremo	Significativo	Considerável	Algum	Pouco	Muito pouco
Número de itens de risco e sua importância	Mais de 10 críticos	5 a 10 críticos	2 a 4 críticos	1 crítico	Mais de 5 não críticos	Menos de 5 não críticos

Nota média	**Muito baixo**	**Baixo**	**Nominal**	**Alto**	**Muito alto**	**Extra-alto**
Interpretação	Pouco (20%)	Algum (40%)	Frequente (60%)	Geralmente (75%)	Largamente (90%)	Totalmente (100%)
Fator numérico	7,07	5,65	4,24	2,83	1,41	0,00

Um risco crítico, ou de alta importância, deve ser considerado como um risco com uma combinação de impacto e probabilidade alta, conforme mostrado na Seção 8.4. Riscos não críticos podem ser os de importância média. Riscos de baixa importância não precisam ser contabilizados na última linha da tabela de RESL.

O fator de escala *TEAM*, ou seja, a *coesão da equipe de desenvolvimento*, pode ser calculado como na Tabela 7-10.

Tabela 7-10: Forma de obtenção do equivalente numérico para TEAM.

Característica	Muito baixo	Baixo	Nominal	Alto	Muito alto	Extra-alto
Consistência dos objetivos e cultura dos interessados	Pouca	Alguma	Básica	Considerável	Forte	Total
Habilidade e vontade dos interessados em acomodar os objetivos de outros interessados	Pouca	Alguma	Básica	Considerável	Forte	Total
Experiência dos interessados em trabalhar como uma equipe	Nenhuma	Pouca	Pouca	Básica	Considerável	Extensiva
Construção de equipes com os interessados para obter visão compartilhada e compromissos	Nenhuma	Pouca	Pouca	Básica	Considerável	Extensiva

Nota média	Muito baixo	Baixo	Nominal	Alto	Muito alto	Extra-alto
Interpretação	Interações muito difíceis	Algumas interações difíceis	Interações basicamente cooperativas	Predominantemente cooperativas	Altamente cooperativas	Interações perfeitas
Fator numérico	5,48	4,38	3,29	2,19	1,10	0,00

Até o momento é possível observar que os fatores de escala do CII não são apenas itens para estimação de esforço, mas também recomendações de boas práticas, ou seja, objetivos a serem buscados. Se cada uma das características listadas obtiverem notas positivas, o tempo de desenvolvimento tenderá a ser muito mais baixo do que com notas mais negativas.

Finalmente o fator *PMAT*, ou *maturidade do processo*, pode ser calculado a partir dos nível de maturidade obtido pela empresa de acordo com o modelo CMM (a Seção 12.3 apresenta o modelo CMMI, sucessor de CMM) ou SPICE (Seção 12.2). O equivalente numérico pode ser obtido como mostrado na Tabela 7-11.

Tabela 7-11: Forma de obtenção do equivalente numérico para PMAT.

Característica	Muito baixo	Baixo	Nominal	Alto	Muito alto	Extra-alto
Nível CMM (ou CMMI)	1 – inferior	1 – superior	2	3	4	5
Nível EPML (ou SPICE)	0	1	2	3	4	5

Nota média	Muito baixo	Baixo	Nominal	Alto	Muito alto	Extra-alto
Interpretação	Sem processo definido	Processo incipiente	Processo definido	Processo gerenciado	Processo padronizado gerenciado quantitativamente	Processo em otimização constante
Fator numérico	7,80	6,24	4,68	3,12	1,56	0,00

Na falta de uma avaliação por CMMI ou SPICE, pode-se aplicar um questionário de avaliação (Boehm B. , 2000) para obter o nível correspondente. Esse valor é correspondente à linha "Nível EPML" na primeira tabela da figura, onde EPML significa "*Estimated Process Maturity Level*". O questionário e formas de cálculo são apresentados na Seção 7.3.5.

Ao contrário das tabelas anteriores, nesta aqui apenas uma opção de avaliação entre CMM/CMMI e EPML/SPICE é necessária, não uma combinação das duas.

7.3.2 Multiplicadores de Esforço

Os multiplicadores de esforço M_i são usados para ajustar a estimativa de esforço para o desenvolvimento de um sistema baseando-se em características próprias do projeto e da equipe que podem onerar este tempo.

CII apresenta dois grupos de multiplicadores de esforço: um para ser aplicado até a fase de elaboração (*Early Design Model*) e outro para ser aplicado depois (*Post-Architecture*) conforme já mostrado na Figura 7-2.

7.3.2.1 Multiplicadores de Esforço do Post-Architecture Model

Os multiplicadores de esforço do *Post-Architecture Model* são divididos nos seguintes grupos:

a) Fatores do Produto.
b) Fatores da Plataforma.
c) Fatores Humanos.
d) Fatores de Projeto.

Os *fatores do produto* avaliam características do produto que possam afetar o esforço de desenvolvimento. Esses fatores são os seguintes:

a) *Software com Confiabilidade Requerida* (RELY).
b) *Tamanho da Base de Dados* (DATA).
c) *Complexidade do Produto* (CPLX).
d) *Desenvolvimento Visando Reusabilidade* (RUSE).
e) *Documentação Necessária para o Ciclo de Desenvolvimento* (DOCU).

O multiplicador de esforço *RELY* (*Software com Confiabilidade Requerida*) avalia o tipo de consequências, caso o software tenha alguma falha. Para aplicar a tabela, deve-se encontrar o descritor (primeira linha) que melhor descreve o fator de confiabilidade requerida. A partir dele encontra-se a avaliação e seu equivalente numérico na coluna correspondente (Tabela 7-12). Por exemplo, se o efeito de uma falha do software for apenas inconveniente, então RELY é muito baixo e o equivalente numérico é 0,82.

Tabela 7-12: Forma de obtenção do equivalente numérico para RELY.

Descritor	Pequena inconveniência	Perdas pequenas, facilmente recuperáveis	Perdas moderadas, facilmente recuperáveis	Alta perda financeira	Risco a vida humana	
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	0,82	0,92	1,00	1,10	1,26	n/a

No caso de RELY, a avaliação “Extra-alto” é inexistente (n/a). Isso também acontecerá com outros multiplicadores, como será visto adiante.

O multiplicador de esforço *DATA* (*Tamanho da Base de Dados*) avalia o tamanho relativo da base de dados usada *para testes* do programa (não a base de dados final). A razão D/P é o número de *Kbytes* na base de dados de teste (*D*) dividido pelo número de milhares de linhas

(*P*) estimado do programa (em KSLOC). A Tabela 7-13 apresenta os parâmetros de cálculo para DATA.

Tabela 7-13: Forma de obtenção do equivalente numérico para DATA.

Descritor		D/P < 10	10 ≤ D/P ≤ 100	100 ≤ D/P ≤ 1000	DP ≥ 1000	
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	n/a	0,90	1,00	1,14	1,28	n/a

O multiplicador de esforço *CPLX* (*Complexidade do Produto*) avalia a complexidade em cinco áreas: operações de controle (estruturas de controle, recursão, concorrência e distribuição), operações computacionais (cálculos matemáticos), operações dependentes de dispositivos (entrada e saída de dados), operações de gerenciamento de dados (desde simples dados em memória até bancos de dados distribuídos) e operações de gerenciamento de interface (simples entrada de texto num extremo e realidade virtual no outro). Então, ao contrário dos multiplicadores anteriores, que são obtidos a partir de um único descritor, CPLX terá cinco descritores (um para cada área). A complexidade do produto é dada pela média subjetivamente ponderada destas cinco áreas conforme a Tabela 7-14.

Tabela 7-14: Forma de obtenção do equivalente numérico para CPLX.

Operações de controle	Código sequencial com poucas estruturas não aninhadas. Composição simples de módulos via chamada de procedimentos ou *script*s	Aninhamento simples de estruturas de controle. Basicamente predicados simples.	Basicamente aninhamento simples. Algum controle intermódulos. Tabelas de decisão. Chamadas ou passagem de mensagens, incluindo processamento distribuído suportado por middleware.	Estruturas altamente aninhadas com vários predicados compostos. Controle de fila e pilha. Processamento distribuído homogêneo. Controle de tempo real simples em processador único.	Código reentrante e recursivo. Gerenciamento de interrupção com prioridade fixa. Sincronização de tarefas. Chamadas complexas. Processamento distribuído heterogêneo. Controle de tempo real complexo em processador único.	Escalonamento de múltiplos recursos com mudança dinâmica de prioridades. Controle em nível de micro código. Controle complexo de tempo real distribuído.
Operações computacionais	Avaliação de expressões simples como A:=B+C*(D-E)	Avaliação de expressões de nível moderado como D:=SQRT(B**2-4.*A*C)	Uso de rotinas matemáticas e estatísticas padrão. Operações básicas sobre matrizes e vetores.	Análise numérica básica: interpolação multivariada e equações diferenciais ordinárias. Arredondamento e truncamento básicos.	Análise numérica complexa, mas estruturada: equações de matrizes, equações diferenciais parciais. Paralelização simples.	Análise numérica complexa e não estruturada: análise de ruído altamente precisa, dados estocásticos. Paralelização complexa.
Operações dependentes de dispositivo	Comandos simples de leitura e escrita com formatação simples.	Sem necessidade de conhecimento de características particulares de processador ou dispositivo de E/S. E/S feita por Get e Put.	Processamento de E/S inclui seleção de dispositivo, checagem de *status* e processamento de erros.	Operações de E/S em nível físico (traduções de endereços de armazenamento físicos; buscas e leituras, etc.). Overlap de E/S otimizado.	Rotinas para diagnóstico de interrupção. Gerenciamento de linha de comunicação. Sistemas embarcados com consideração intensiva de performance.	Codificação de dispositivos dependentes de tempo. Operações microprogramadas. Sistemas embutidos com performance crítica.
Operações de gerenciamento de dados	Arrays simples em memória. Simples consultas e atualizações em COTS ou banco de dados.	Arquivos simples sem edição nem buffers. Consultas e atualizações em bancos de dados ou COTS moderadamente complexas.	Entrada de múltiplos arquivos e saída em arquivo único. Mudanças estruturais simples. Edição simples. Consultas e atualizações complexas em COTS ou banco de dados.	Gatilhos simples ativados pelo conteúdo de sequências de dados. Reestruturação de dados complexa.	Coordenação de bancos de dados distribuídos. Gatilhos complexos. Otimização.	Estruturas relacionais e de objetos dinâmicas e altamente acopladas. Gerenciamento de dados em linguagem natural.
Operações de gerenciamento de interface com usuário	Formulários de entrada simples e geradores de relatórios.	Uso de construtores de interface com usuário (GUI) simples	Simples uso de um conjunto de *widgets*.	Desenvolvimento e extensão de conjunto de widgets. E/S por voz. Multimídia.	Gráficos dinâmicos 2D e 3D moderadamente complexos. Multimídia.	Multimídia complexa. Realidade virtual. Interface em linguagem natural.
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	0,73	0,87	1,00	1,17	1,34	1,74

O multiplicador de esforço *RUSE* (*Desenvolvimento Visando Reusabilidade*) avalia o quanto o projeto é feito pensando em gerar componentes que possam ser depois reusados. O

desenvolvimento baseado em SPL (Linhas de Produto de Software - Seção 3.14), por exemplo, leva a um valor alto de RUSE. A Tabela 7-15 apresenta o padrão de atribuição de notas a este multiplicador.

Tabela 7-15: Forma de obtenção do equivalente numérico para RUSE.

Descritor		Nenhum reuso	Dentro do projeto	Dentro de um programa	Dentro de uma *SPL*	Entre múltiplas *SPLs*
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	n/a	0,95	1,00	1,07	1,15	1,24

O multiplicador *DOCU* (*Documentação Necessária para o Ciclo de Desenvolvimento*) mede o quanto a documentação necessária para o desenvolvimento realmente é produzida. Se não há compromisso com a documentação DOCU é baixo; se há documentação em excesso, além das necessidades reais então o índice é alto. A Tabela 7-16 mostra como chegar aos valores numéricos deste multiplicador.

Tabela 7-16: Forma de obtenção do equivalente numérico para DOCU.

Descritor	Muitas necessidades de ciclo de vida não cobertas	Algumas necessidades de ciclo de vida não cobertas	Exatamente dimensionada para as necessidades do ciclo de vida	Excessiva para as necessidades do ciclo de vida	Muito excessiva para as necessidades do ciclo de vida	
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	0,81	0,91	1,00	1,11	1,23	n/a

Os multiplicadores de esforço referentes à *plataforma* se referem à complexidade da plataforma alvo de implementação (hardware e software básico). Estes fatores são os seguintes:

a) *Restrição de Tempo de Execução* (TIME).
b) *Restrição de Memória Principal* (STOR).
c) *Volatilidade da Plataforma* (PVOL).

O multiplicador *TIME* (*Restrição de Tempo de Execução*) avalia a porcentagem esperada de uso dos processadores disponíveis pela aplicação (Tabela 7-17).

Tabela 7-17: Forma de obtenção do equivalente numérico para TIME.

Descritor			Menos de 50% de uso do tempo de execução disponível	70% de uso do tempo de execução disponível	85% de uso do tempo de execução disponível	95% de uso do tempo de execução disponível.
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	n/a	n/a	1,00	1,11	1,29	1,63

O multiplicador *STOR* (*Restrição de Memória Principal*) avalia a porcentagem esperada de uso da memória principal pela aplicação. A Tabela 7-18 apresenta os equivalentes numéricos para este multiplicador.

Tabela 7-18: Forma de obtenção do equivalente numérico para STOR.

Descritor			Menos de 50% de uso da memória principal	70% de uso da memória principal	85% de uso da memória principal	95% de uso da memória principal
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	n/a	n/a	1,00	1,05	1,17	1,46

O uso de técnicas como *swap* nos modernos sistemas operacionais pode fazer com que a preocupação com o uso da memória principal seja sempre nominal no caso de sistemas de informação, porque a memória sempre poderá ser aumentada virtualmente. Porém, em aplicações embarcadas esse indicador pode ser avaliado de forma mais crítica justamente pela falta deste tipo de mecanismo.

O multiplicador *PVOL* (*Volatilidade da Plataforma*) avalia a plataforma de desenvolvimento, a qual inclui o hardware e software básico sobre os quais a aplicação é construída. O fator é avaliado como baixo quando ocorrem mudanças de plataforma em períodos superiores a um ano e alto quando as mudanças ocorrem em média a cada duas semanas. A Tabela 7-19 mostra como obter os valores numéricos.

Tabela 7-19: Forma de obtenção do equivalente numérico para PVOL.

Descritor		Mudanças grandes a cada 12 meses, pequenas a cada mês	Grandes: 6 meses; pequenas: 2 semanas	Grandes: 2 meses; pequenas: 1 semana	Grandes: 2 semanas; pequenas: 2 dias	
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	n/a	0,87	1,00	1,15	1,30	n/a

Os *fatores humanos* considerados como multiplicadores de esforço são os seguintes:

a) *Capacidade dos Analistas* (ACAP).
b) *Capacidade dos Programadores* (PCAP).
c) *Continuidade de Pessoal* (PCON).
d) *Experiência em Aplicações Semelhantes* (APEX).
e) *Experiência na Plataforma* (PLEX).
f) *Experiência na Linguagem e Ferramentas* (LTEX).

O multiplicador *ACAP* (*Capacidade dos Analistas*) avalia a capacidade dos analistas em analisar e modelar aplicações, eficiência e eficácia, e habilidade de cooperar e comunicar. Quanto maior a capacidade, menor o valor de ACAP. A Tabela 7-20 mostra como obter o valor numérico para a capacidade dos analistas. Esta é avaliada em termos de percentis. Por

exemplo, se os analistas estão no percentil 15% mais baixo então o multiplicador é avaliado como “Muito baixo”.

Tabela 7-20: Forma de obtenção do equivalente numérico para ACAP.

Descritor	Percentil 15	35	55	75	90	
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	1,42	1,19	1,00	0,85	0,71	n/a

O multiplicador *PCAP* (*Capacidade dos Programadores*) avalia os programadores de forma semelhante a ACAP (Tabela 7-21).

Tabela 7-21: Forma de obtenção do equivalente numérico para PCAP.

Descritor	Percentil 15	35	55	75	90	
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	1,34	1,15	1,00	0,88	0,76	n/a

O multiplicador *PCON* (*Continuidade de Pessoal*) avalia a percentagem de trocas de desenvolvedores no período de um ano. Quanto menos trocas, menor o valor de PCON. A Tabela 7-22 mostra como obter o valor numérico de PCON a partir da porcentagem de troca de desenvolvedores no período de um ano.

Tabela 7-22: Forma de obtenção do equivalente numérico para PCON.

Descritor	48%/ano	24%/ano	12%/ano	6%/ano	3%/ano	
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	1,29	1,12	1,00	0,90	0,81	n/a

O multiplicador *APEX* (*Experiência em Aplicações Semelhantes*) avalia o tempo médio (em anos) de experiência da equipe em aplicações semelhantes à que vai ser desenvolvida. Quanto maior o tempo, menor o valor de APEX. A Tabela 7-23 mostra como obter os valores para APEX.

Tabela 7-23: Forma de obtenção do equivalente numérico para APEX.

Descritor	Menos de 2 meses	6 meses	1 ano	3 anos	6 anos	
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	1,22	1,10	1,00	0,88	0,81	n/a

O multiplicador *PLEX* (*Experiência na Plataforma*) avalia a experiência da equipe na plataforma de desenvolvimento, incluindo bibliotecas, hardware, sistema operacional, banco de dados, *middleware* e outros itens relacionados. A Tabela 7-24 apresenta a forma de obter os equivalentes numéricos para PLEX.

Tabela 7-24: Forma de obtenção do equivalente numérico para PLEX.

Descritor	Menos de 2 meses	6 meses	1 ano	3 anos	6 anos	
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	1,19	1,09	1,00	0,91	0,85	n/a

O multiplicador *LTEX* (*Experiência na Linguagem e Ferramentas*) avalia o tempo médio de experiência da equipe nas ferramentas CASE e linguagens usadas para o desenvolvimento (Tabela 7-25).

Tabela 7-25: Forma de obtenção do equivalente numérico para LTEX.

Descritor	Menos de 2 meses	6 meses	1 ano	3 anos	6 anos	
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	1,20	1,09	1,00	0,91	0,84	n/a

Os *fatores de projeto* avaliam a influência do uso de ferramentas modernas de desenvolvimento, ambiente de trabalho e aperto do cronograma. Estes fatores são:

a) *Uso de Ferramentas de Software* (TOOL).
b) *Equipe de Desenvolvimento Distribuída* (SITE).
c) *Cronograma de Desenvolvimento Requerido* (SCED).

O multiplicador *TOOL (Uso de Ferramentas de Software*) avalia a qualidade do suporte computacional ao ambiente de desenvolvimento. O uso de simples compiladores implica em um índice ruim (alto) para este fator, enquanto que o uso ferramentas CASE e de gerenciamento de projeto que integram todas as atividades de desenvolvimento implicam em uma boa avaliação (índice baixo). A Tabela 7-26 mostra como obter os valores numéricos para TOOL.

Tabela 7-26: Forma de obtenção do equivalente numérico para TOOL.

Descritor	Editar, codificar, debugar.	CASE simples. Pouca integração.	Ferramentas básicas de ciclo de vida moderadamente integradas.	Ferramentas de ciclo de vida fortes e maduras, moderadamente integradas	Ferramentas de ciclo de vida fortes, maduras e bem integradas com processos, métodos e reuso	
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	1,17	1,09	1,00	0,90	0,78	n/a

O multiplicador *SITE* (*Equipe de Desenvolvimento Distribuída*) avalia a influência da distribuição da equipe de desenvolvimento. Equipes distribuídas internacionalmente apontam para um aumento de carga em relação a este fator, enquanto que uma equipe que trabalha toda na mesma sala implica em uma carga menor. Ao contrário da maioria dos multiplicadores, SITE tem dois descritores. Deve ser feita a média subjetiva entre os dois para determinar a nota do multiplicador. A Tabela 7-27 mostra como obter o valor numérico para SITE.

Tabela 7-27: Forma de obtenção do equivalente numérico para SITE.

Descritor de co-locação	Internacional	Multi-cidade e multi-empresa	Multi-cidade ou multi-empresa	Mesma cidade ou área metropolitana	Mesmo edifício ou complexo	Totalmente co-locada
Descritor de comunicação	Alguns telefones, correio	Telefones individuais, FAX	Email	Comunicação eletrônica de banda larga	Videoconferência	Multimídia interativa
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	1,22	1,09	1,00	0,93	0,86	0,80

Então, por exemplo, uma equipe internacional (muito baixo) que utilize videoconferência (muito alto) pode ser avaliada na média destes dois descritores: nominal.

O multiplicador *SCED* (*Cronograma de Desenvolvimento Requerido*) reflete o grau requerido de aceleração forçada a um cronograma nominal ideal ou seu relaxamento (Tabela 7-28). Um cronograma forçadamente mais rápido do que o usual implica em um fator com valor mais alto, ou seja, maior esforço de desenvolvimento.

Tabela 7-28: Forma de obtenção do equivalente numérico para SCED.

Descritor	75% do tempo nominal	85%	100%	130%	160%	
Avaliação	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	1,43	1,14	1,00	1,00	1,00	n/a

7.3.2.2 Multiplicadores de Esforço do Early Design Model

Os multiplicadores de esforço do *Early Design Model* são o resultado de combinações dos fatores do *Post-Architecture Model*, além de outras informações. Parte-se do princípio de que quando este modelo for aplicado ainda não se tem muita informação sobre o real ambiente de desenvolvimento e características do projeto. Dessa forma, as avaliações dos 17 multiplicadores de esforço do *Post Architecture Model* serão meras estimativas. Assim, o *Early Design Model* cria seus multiplicadores de esforço a partir de uma combinação destas estimativas com outras informações mais provavelmente conhecidas nesta fase de um projeto.

Os fatores multiplicadores de esforço do *Early Design Model* são os seguintes:

a) *Capacidade de Pessoal* (PERS).
b) *Confiabilidade e Complexidade do Produto* (RCPX).
c) *Desenvolvimento para Reuso* (RUSE).
d) *Dificuldade com a Plataforma* (PDIF).
e) *Experiência do Pessoal* (PREX).
f) *Instalações* (FCIL).
g) Cronograma de Desenvolvimento Requerido (SCED).

O multiplicador *PERS (Capacidade de Pessoal*) é obtido a partir da média subjetiva de três descritores. O primeiro é a soma dos multiplicadores ACAP, PCAP e PCON, conforme definidos para a fase e construção, mas considerando a seguinte equivalência numérica:

a) Muito baixo = 1.

b) Baixo = 2.
c) Nominal = 3.
d) Alto = 4.
e) Muito alto = 5.
f) Extra-alto = 6.

Assim, o valor da soma dos três multiplicadores, que neste caso variam de muito baixo (1) a muito alto (5) dará um resultado entre 3 e 15. O segundo descritor é o percentil combinado de ACAP e PCAP. O terceiro é o percentual anual de troca de pessoal. Note-se que aqui passa a existir a nota "Extrabaixo" que não havia nos multiplicadores Post-Architecture. A Tabela 7-29 mostra como obter os valores para PERS.

Tabela 7-29: Forma de obtenção do equivalente numérico para PERS.

Soma de ACAP, PCAP e PCON	3 a 4	5 a 6	7 a 8	9	10 a 11	12 a 13	14 a 15
Média dos percentis ACAP e PCAP	20%	35%	45%	55%	65%	75%	85%
Taxa de troca de pessoal anual	45%	30%	20%	12%	9%	6%	4%
Avaliação	Extra baixo	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	2,12	1,62	1,26	1,00	0,83	0,63	0,50

O multiplicador *RCPX* (*Confiabilidade e Complexidade do Produto*) é uma combinação dos fatores RELY, DATA, CPLX e DOCU. Ele é obtido a partir da média subjetiva de quatro descritores. A Tabela 7-30 mostra como obter os equivalentes numéricos para este multiplicador.

Tabela 7-30: Forma de obtenção do equivalente numérico para RCPX.

Soma de RELY, DATA, CPLX e DOCU	5 a 6	7 a 8	9 a 11	12	13 a 15	16 a 18	19 a 21
Ênfase em confiabilidade e documentação	Muito pouca	Pouca	Alguma	Básica	Forte	Muito forte	Extrema
Complexidade do produto	Muito simples	Simples	Alguma	Moderada	Complexa	Muito complexa	Extremamente complexa
Tamanho do banco de dados	Pequeno	Pequeno	Pequeno	Moderado	Grande	Muito grande	Muito grande
Avaliação	Extra baixo	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	0,49	0,60	0,83	1,00	1,33	1,91	2,72

O multiplicador *RUSE* (*Desenvolvimento para Reuso*) nesta fase é exatamente o mesmo que foi calculado para *Post-Architecture*.

O multiplicador *PDIF* (*Dificuldade com a Plataforma*) é uma combinação de TIME, STOR e PVOL e mais dois descritores. A Tabela 7-31 mostra como obter os valores para PDIF.

Tabela 7-31: Forma de obtenção do equivalente numérico para PDIF.

Soma de TIME, STOR e PVOL			8	9	10 a 12	13 a 15	16 a 17
Restrição de tempo e memória combinadas de TIME e STOR			Menos de 50%	Menos de 50%	65%	80%	90%
Volatilidade da plataforma			Muito estável	Estável	Um tanto volátil	Volátil	Altamente volátil
Avaliação	Extra baixo	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	n/a	n/a	0,87	1,00	1,29	1,81	2,61

O multiplicador *PREX (Experiência do Pessoal*) é uma combinação de APEX, LTEX e PLEX combinando com o descritor que refere a experiência média da equipe. A Tabela 7-32 mostra como obter os valores para PREX.

Tabela 7-32: Forma de obtenção do equivalente numérico para PREX.

Soma de APEX, PLEX e LTEX	3 a 4	5 a 6	7 a 8	9	10 a 11	12 a 13	14 a 15
Experiência média nas aplicações, plataforma, linguagem e ferramentas	Menos de 3 meses	5 meses	9 meses	1 ano	2 anos	4 anos	6 anos
Avaliação	Extra baixo	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	1,59	1,33	1,22	1,00	0,87	0,74	0,62

O multiplicador *FCIL* (*Instalações*) é uma combinação de TOOL e SITE, além de mais 2 descritores. A Tabela 7-33 mostra como obter os valores para FCIL.

Tabela 7-33: Forma de obtenção do equivalente numérico para FCIL.

Soma de TOOL e SITE	2	3	4 a 5	6	7 a 8	9 a 10	11
Suporte por ferramentas	Mínimo	Algum	Coleção simples de ferramentas CASE	Ferramentas básicas de ciclo de vida	Bom, moderadamente integrado	Forte, moderadamente integrado	Forte, bem integrado
Condições multisite	Suporte fraco em ambiente multisite complexo	Algum suporte para desenvolvimento multisite complexo	Algum suporte para desenvolvimento multisite moderadamente complexo	Suporte básico para ambiente multisite moderadamente complexo	Suporte forte para ambiente multisite moderadamente complexo	Suporte forte para ambiente multisite simples	Suporte muito forte para equipe co-locada ou ambiente multisite simples
Avaliação	Extra baixo	Muito baixo	Baixo	Nominal	Alto	Muito Alto	Extra-alto
Equivalente numérico	1,43	1,30	1,10	1,00	0,87	0,73	0,62

O multiplicador *SCED* (*Cronograma de Desenvolvimento Requerido*) é o mesmo SCED calculado para *Post-Architecture*.

7.3.3 Aplicando COCOMO II para as fases do UP

Conforme mostrado na Tabela 7-34, CII estima o esforço *E* a ser despendido nas fases de Elaboração e Construção. Considera-se assim que 100% do valor *E* estará contido nestas fases.

O esforço despedido nas fases de Concepção de Transição será *somado* a este valor, passando assim de 100%. O mesmo raciocínio se aplica ao tempo linear *T.* As colunas "intervalo" na Tabela 7-34 indicam os limites dentro dos quais espera-se que o esforço e o tempo linear possam variar.

Tabela 7-34: Aplicação de esforço e tempo linear às fases do UP.

Fase	Esforço nominal	Intervalo	Tempo linear nominal	Intervalo
Concepção	0,06*E*	0,02 a 0,15	0,125*T*	0,02 a 0,3
Elaboração	0,24*E*	0,20 a 0,28	0,375*T*	0,33 a 0,42
Construção	0,76*E*	0,72 a 0,80	0,625*T*	0,58 a 0,67
Transição	0,12*E*	0,00 a 0,20	0,125*T*	0,00 a 0,20
Totais	**1,18*E***		**1,25*T***	

Essa tabela é compatível com a Figura 6-1, onde os valores de esforço e tempo linear são convertidos em percentagens do esforço total das quatro fases.

Assim, por exemplo, um projeto com E = 56 desenvolvedor-mês e T = 11,5 meses terá, em média os valores de esforço (em desenvolvedor-mês) e duração por fase definidos como na Tabela 7-35.

Tabela 7-35: Exemplo de cálculo de tempo e esforço para as fases do UP de um projeto com *E* = 56 e *T* = 11,5.

Fase	Esforço (desenvolvedor-mês)	Tempo (meses)
Concepção	3,4	1,4
Elaboração	13,4	4,3
Construção	42,6	7,2
Transição	6,7	1,4
Totais	**66,1**	**14,3**

Os tempos e esforços por fase, porém, variam dentro de faixas, conforme mostrado na Tabela 7-34. O esforço e tempo das fases de Elaboração e Construção devem variar inversamente, porque ambos somados devem resultar sempre no valor *E* e *T* respectivamente.

Mas as fases de Concepção e Transição podem variar livremente dentro dos intervalos definidos. Seguem alguns exemplos de fatores que podem aumentar ou diminuir a duração destas fases:

a) Requisitos bem definidos já no início do projeto (por exemplo, criar um substituto para um sistema que já existe) fazem com que a fase de Concepção seja bem menor (em tempo e esforço) do que seu valor nominal.
b) Se o sistema, depois de pronto, vai mudar a maneira como as pessoas trabalham, então a fase de Transição deverá ser bem maior.
c) Se existem importantes riscos técnicos, a fase de Concepção será maior.
d) Se a comunidade de usuários é grande e heterogênea, a fase de Concepção devera ser maior.
e) Se houver necessidade de integração com hardware e sistemas legados existentes, então a fase de Transição será maior.

Em resumo, mais incerteza em relação aos requisitos implica em uma fase de Concepção maior do que o valor nominal. E implantações complexas do sistema implicam em uma fase de Transição maior.

O manual de CII (Boehm B. , 2000)[87] ainda apresenta outras informações como, por exemplo, como estimar a duração relativa de cada disciplina e atividade do UP nas diferentes fases, o que é resumido na Tabela 7-36.

Tabela 7-36: Resumo do esforço relativo às disciplinas UP nas diferentes fases.

Disciplina	Concepção	Elaboração	Construção	Transição
Gerenciamento	14%	12%	10%	14%
Ambiente/Configuração	10%	8%	5%	5%
Requisitos	38%	18%	8%	4%
Design	19%	36%	16%	4%
Implementação	8%	13%	34%	19%
Avaliação/Teste	8%	10%	24%	24%
Implantação	3%	3%	3%	30%
Total	**100%**	**100%**	**100%**	**100%**

7.3.4 Calibragem do Modelo

Os valores das constantes de CII foram definidas a partir de um conjunto de projetos base. Para obter melhores resultados em uma empresa específica é necessário que essas constantes sejam calibradas para os parâmetros específicos da empresa. Além disso, com o passar do tempo esses valores poderão mudar também, exigindo novas calibragens.

A constante *A*, por exemplo, na equação geral de estimação de esforço tem seu valor definido em 2,94:

$$E = A * KSLOC^S * \prod_{i=1}^{n} M_i$$

Para obter um valor mais adequado ao ambiente local de trabalho recomenda-se que se tenha realizado pelo menos 5 projetos, para os quais se tenha a estimativa e o valor real de esforço realizado. A Tabela 7-37, adaptada de Boehm (2000), mostra os dados obtidos para oito projetos.

[87] csse.usc.edu/csse/research/COCOMOII/cocomo2000.0/CII_modelman2000.0.pdf

Tabela 7-37: Exemplo de calibragem para a constante *A*.

Real	$KSLOC^S * \prod_{i=1}^{n} M_i$	*ln*(*Real*)	$ln(KSLOC^S * \prod_{i=1}^{n} M_i)$	$ln(Real)\text{-}ln(KSLOC^S * \prod_{i=1}^{n} M_i)$
1854,6	686,7	7,53	6,53	0,99
258,5	94,3	5,55	4,55	1,01
201,0	77,7	5,30	4,35	0,95
58,9	20,3	4,08	3,01	1,07
9661,0	3338,8	9,18	8,11	1,06
7021,3	2753,5	8,86	7,92	0,94
91,7	38,9	4,52	3,66	0,86
689,7	301,1	6,54	5,71	0,83
				X = 0,96
				A = 2,62

A primeira coluna (*Real*) mostra o esforço real de cada projeto em desenvolvedor-mês (fases de Elaboração e Construção). A segunda coluna apresenta o cálculo da estimativa não ajustada, ou seja, a fórmula do esforço total sem a constante *A*:

$$KSLOC^S * \prod_{i=1}^{n} M_i.$$

Esses são os valores básicos para o cálculo.

As duas colunas seguintes apresentam o logaritmo natural (*ln*) do valor real do esforço e da estimativa não ajustada. Finalmente, a última coluna apresenta a diferença entre essas duas colunas.

No final da tabela, o valor *X* corresponde à média das diferenças (última coluna). A constante *A* é calculada como sendo o anti-logaritmo dessa média, ou seja, $A = e^X$. Esse exemplo mostra que neste ambiente local a constante *A* deveria ser 2,62 e não 2,94.

Boehm (2000) também mostra como calibrar a distribuição das estimativas por atividade e fase, para o ambiente local, o que pode ser particularmente interessante quando essas estimativas são efetivamente usadas para calcular esforço dentro dos ciclos iterativos.

7.3.5 Questionário EPML

O questionário *EPML* (*Estimated Process Matury Level*) é uma interessante ferramenta não só para avaliar o fator PMAT, como também como uma auto-avaliação da empresa em relação às boas práticas de processo. Principalmente por essa segunda razão ele é reproduzido aqui nesta seção.

O questionário se subdivide em 18 áreas chave correspondentes às áreas de avaliação de processo do CMM. Cada pergunta deve ser respondida com uma das opções:

a) *Quase sempre*. Quando os objetivos são consistentemente obtidos e bem estabelecidos em procedimentos operacionais padrão (mais de 90% das vezes).
b) *Frequentemente*. Quando os objetivos são obtidos com relativa frequência, mas algumas vezes são omitidos por conta de circunstâncias difíceis (entre 60 e 90% das vezes).

c) *Metade do tempo*. Quando os objetivos são obtidos em cerca de metade das vezes (de 40 a 60% das vezes).
d) *Ocasionalmente*. O quando os objetivos são obtidos algumas vezes, mas com pouca frequência (entre 10 a 40% das vezes).
e) *Raramente*. Quando os objetivos raramente ou nunca são obtidos (menos de 10% das vezes).
f) *Não se aplica*. Quando se tem o conhecimento necessário sobre a organização, o projeto e a área-chave, mas se entende que área-chave não se aplica nas circunstâncias.
g) *Não sabe*. Quando não se sabe o que responder a para a área chave.

Deve ser dada uma resposta para cada uma das 18 áreas-chave.

a) *Gerenciamento de requisitos* (Os requisitos do software são controlados a ponto de estabelecer uma *baseline* para uso da gerência de engenharia de software? Os planos, produtos e atividades relacionados ao software são mantidos consistentes com os requisitos do software?).
b) *Planejamento de projeto de software* (Estimativas de esforço são documentadas para uso em planejamento e rastreamento de projeto de software? As atividades são planejadas e documentadas? Grupos afetados e indivíduos concordam formalmente com seus entendimentos (*commit*ments) relacionados ao projeto de software?).
c) *Rastreamento e supervisão de projeto de software* (Resultados de desempenho reais são rastreados em relação aos planos de software? Ações corretivas são tomadas e gerenciadas até o final, quando os resultados e desempenho reais se desviam significativamente dos planos de software? Mudanças nos entendimentos do software são acordadas formalmente pelos grupos ou indivíduos afetados?).
d) *Gerenciamento de subcontratos de software* (O contratador seleciona subcontratados qualificados? O contratador e o subcontratado concordam formalmente com seus entendimentos um com outro? O contratador e o subcontratado mantêm comunicações constantes? O contratador rastreia os resultados e desempenho reais do subcontratado em relação aos seus empreendimentos?).
e) *Garantia de qualidade de software* (As atividades de garantia de qualidade de software são planejadas? Aderência de produtos e atividades de software aos padrões, procedimentos e requisitos aplicáveis é verificada objetivamente? Grupos e indivíduos afetados são informados das atividades e resultados de garantia de qualidade de software? Assuntos de não conformação que não podem ser resolvidos dentro do projeto de software são levados a gerência superior?).
f) *Gerenciamento de configuração de software* (As atividades de gerenciamento de configuração de software são planejadas? Os produtos de trabalho selecionados são identificados, controlados e disponibilizados? Mudanças nos produtos de trabalho identificadas são controladas? Os grupos e indivíduos afetados são informados do *status* e conteúdo das *baselines* de software?).
g) *Foco do processo organizacional* (Atividades de desenvolvimento e melhoria de processo de software são coordenadas entre as diferentes partes da organização? Os pontos fortes e fracos do processo de software usado são identificados relativamente

a um processo padrão? A atividades de desenvolvimento e melhoria de processo em nível organizacional são planejadas?).

h) *Definição de processo organizacional* (Um processo de software padrão para a organização é desenvolvido e mantido? Informação relacionada ao uso do processo de software padrão pelos projetos é coletada, revisada e disponibilizada?).
i) *Programa de treinamento* (Atividades de treinamento são planejadas? Treinamento para desenvolvimento de habilidades e conhecimentos necessários para realizar a gerência e os papéis técnicos na área de software é fornecido? Os indivíduos no grupo de engenharia de software e grupos relacionados recebem o treinamento necessário para realizar os seus papéis?).
j) *Gerenciamento integrado de software* (O processo de software definido para o projeto é uma versão especializada do processo de software padrão da organização? O projeto é planejado e gerenciado de acordo com o processo de software definido para ele?).
k) *Engenharia de produto de software* (As atividades de engenharia de software são definidas, integradas e consistentemente realizadas para produzir o software? Os produtos de trabalho são mantidos consistentes uns com os outros?).
l) *Coordenação intergrupos* (Os requisitos do usuário são acordados por todos os grupos afetados? Os entendimentos entre os grupos de engenharia são acordados pelos outros grupos afetados? Os grupos de engenharia identificam, rastreiam e resolvem assuntos inter grupos?).
m) *Revisões* (Atividades de revisão são planejadas? Defeitos nos produtos de trabalho são identificados e removidos?).
n) *Gerenciamento quantitativo de processo* (As atividades de gerenciamento quantitativo de processo são planejadas? O desempenho de processo para o projeto definido é controlado quantitativamente? A capacidade do processo de software padrão da organização é conhecida em termos quantitativos?).
o) *Gerenciamento da qualidade de software* (As atividades de gerenciamento de qualidade do projeto de software são planejadas? Objetivos mensuráveis para a qualidade de produto de software e suas prioridades são definidos? O progresso real na direção de obter as metas de qualidade para os produtos de software é quantificado e gerenciado?).
p) *Prevenção de defeitos* (Atividades de prevenção de defeitos são planejadas? Causas comuns de efeitos são detectadas e identificadas? Causas comuns de defeitos são priorizadas e sistematicamente eliminadas?).
q) *Gerenciamento de mudança tecnológica* (A incorporação de mudanças na tecnologia é planejada? Novas tecnologias são avaliadas para determinar o seu efeito na qualidade e produtividade? Novas tecnologias apropriadas são transferidas e praticadas normalmente por toda a organização?).
r) *Gerenciamento de mudança de processo* (Melhoria de processo contínua é planejada? Toda a organização participa das atividades de melhoria de processo de software? O processo de software padrão da organização e os projetos definidos são continuamente melhorados?

Para obter o valor de EPML (entre 0 e 5) que vai permitir avaliar o multiplicador PMAT é necessário transformar as respostas acima em um número. Inicialmente elimina-se áreas-

chave para as quais a resposta foi “não se aplica” ou “não sei”. Fica-se então com *n* respostas válidas. Apenas deve-se tomar cuidado porque quanto menor o *n* menos confiança se poderá ter no resultado da avaliação. A princípio todas as 18 áreas deveriam ter sua avaliação feita.

As respostas dadas são convertidas em números K_i (para $1 \leq i \leq n$) da seguinte maneira:

a) *Quase sempre*: 1,00.
b) *Frequentemente*: 0,75.
c) *Metade do tempo*: 0,50.
d) *Ocasionalmente*: 0,25.
e) *Raramente*: 0,01.

Então EPML é calculado através da seguinte fórmula:

$$EPML = 5 * \frac{(\sum_{i=1}^{n} K_i)}{n}$$

Finalmente o valor é arredondado para o valor inteiro mais próximo e aplicado na Tabela 7-11 para obter a avaliação de PMAT.

7.4 Pontos de Função

A técnica de *Análise Pontos de Função* (*APF*), ou *Function Point Analysis* (Albrecht & Gaffney Jr., 1983) [88] é também uma técnica paramétrica para estimação de esforço para desenvolvimento de software. Porém, ao contrário de COCOMO, ela não se baseia em linhas de código, mas em requisitos.

A análise de pontos de função é aplicável, portanto, a partir do momento em que os requisitos funcionais do software tenham sido definidos. Esses requisitos serão convertidos em valores numéricos, que depois de ajustados à capacidade da empresa desenvolvedora, representarão o esforço necessário para desenvolver o sistema. Assim, a medida obtida pela técnica é, a princípio, independente de linguagem de programação e de tecnologia empregada.

A APF pode ser aplicada para medir o tamanho de um sistema, antes de desenvolvê-lo, de forma que seu custo possa ser mais adequadamente previsto. Além disso, ela pode ser aplicada também a processos de manutenção, permitindo estimar o esforço necessário para implantar uma determinada alteração no software, especialmente se for uma adição de nova funcionalidade.

A técnica também pode ser usada para calcular o custo-benefício de software ou componentes de software comprados, uma vez que a divisão do preço do produto pelo número de pontos de função que ele implementa dá uma ideia de custo relativo do produto. Então, o custo por ponto de função pode ser visto como um fator de normalização para a comparação de produtos de software.

A técnica, como é baseada em requisitos implementados, e não no número de linhas de código produzidas, é mais adequada para medir a produtividade de uma equipe de desenvolvimento,

[88] ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1703110

pois ferramentas de produtividade e reusabilidade permitirão implementar mais funcionalidades perceptíveis para o cliente com menos linhas de código.

Existem três contagens específicas de pontos de função:

a) *Contagem para desenvolvimento de projeto*. Esta técnica é usada para estimar o esforço para o desenvolvimento de um novo projeto.
b) *Contagem para melhoria de projeto* (Seção 14.5.8). Esta técnica é usada para evolução de software, onde se conta as funcionalidades adicionadas, alteradas e removidas. A técnica é aplicável apenas para a manutenção adaptativa, já que a manutenção corretiva e perfectiva são muito imprevisíveis (Seção 14.2).
c) *Contagem de aplicação*. Esta técnica é usada para contar pontos de função de aplicações existentes. Essa contagem pode ter vários objetivos, entre os principais, estimar o tamanho funcional da aplicação, de forma a relativizar outras métricas (Seção 9.5). Por exemplo, pode ser mais realista conhecer o número de defeitos por ponto de função do que simplesmente o número de defeitos do software.

A contagem de pontos de função segue um método composto por seis passos que serão detalhadamente explicados mais adiante no texto:

a) Determinar o tipo de contagem (desenvolvimento, melhoria ou aplicação existente).
b) Determinar os limites da aplicação (escopo do sistema).
c) Identificar e atribuir valor em pontos de função não ajustados para as transações sobre dados (entradas, consultas e saídas externas).
d) Identificar e atribuir valor em pontos de função não ajustados (UFP) para os dados estáticos (arquivos internos e externos).
e) Determinar o fator de ajuste técnico (VAF).
f) Calcular o número de pontos de função ajustados (AFP).

Após este passo ainda é possível obter o esforço e custo total do projeto, sua duração linear e tamanho ideal de equipe.

A técnica é divulgada e normatizada internacionalmente pelo *IFPUG* (*International Function Point Users Group*)[89] e no Brasil pelo *BFPUG* (*Brazilian Function Point Users Group*)[90]. A técnica é reconhecida como métrica de software pela ISO na norma ISO/IEC 20926 – *Software Engineering – Funcion Point Counting Practices Manual*[91].

Além do método de contagem do IFPUG, apresentado nas seções seguintes, existem outros dois métodos internacionalmente relevantes: NESMA[92], da associação holandesa de métricas, e Mark II (Symons, 1988)[93], ou MK II, mantido pela associação inglesa de métricas. Ao contrário do manual de contagem do IFPUG, que deve ser adquirido, os manuais destas duas técnicas podem ser obtidos gratuitamente em seus *sites*, bastando fazer registro gratuito na respectiva associação.

[89] www.ifpug.org/
[90] www.bfpug.com.br/
[91] www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=35582
[92] www.nesma.nl/section/home/
[93] www.ceng.metu.edu.tr/~e120353/HoughTransform/FP_analysis.pdf

7.4.1 Interpretação e Classificação dos Requisitos como Funções

A APF é baseada na contagem de pontos para cada uma das funções do sistema. Primeiramente deve-se ter em mente que apenas funcionalidades visíveis para o usuário devem ser contadas. Então não é todo e qualquer requisito que conta. Se um requisito menciona algum cálculo interno apenas, ele não deve ser contado como função, embora possivelmente vá aparecer como parte de outra função.

Apenas transferências de informação para dentro e para fora do escopo do sistema (e arquivos de dados mantidos no sistema e acessíveis pelo usuário) são considerados funções.

Outra coisa é que um requisito pode ter mais do que uma função representada nele. Se o requisito fala, por exemplo, em manter um cadastro de clientes e de produtos, então são, na verdade, dois cadastros, que devem ser interpretados como dois arquivos individuais, cada um com sua contagem de pontos.

Assim, um primeiro passo para o uso da técnica é tomar os requisitos, eliminar aqueles que são meramente funções internas (ou incorporá-los às funções a que eventualmente pertençam), e subdividir aqueles que representam mais do que uma função, até que se fique com uma lista de funções individuais que sejam visíveis para o usuário.

Embora trabalhos como o de Longstreet (2012)[94] procurem interpretar essas funções em termos de interface com usuário, o ideal é que sejam identificadas nos requisitos funcionais, ou seja, na sua forma mais essencial e independente de tecnologia.

A técnica APF avalia as duas naturezas dos dados:

a) *Dados estáticos*, ou seja, a representação estrutural dos dados, na forma de arquivos internos ou externos.
b) *Dados dinâmicos*, ou seja, a representação das transações sobre os dados, na forma em entradas, saídas e consultas externas.

As transações (entradas, saídas e consultas externas) devem ser processos elementares, isso é, de único passo. Uma transação é a menor unidade de atividade que faça sentido do ponto de vista do usuário e (importante!) deixe o sistema em um estado consistente.

Por outro lado, os arquivos internos ou externos devem ser elementos complexos de informação reconhecíveis pelo usuário (ou seja, não são necessariamente representações físicas internas). Podem ser considerados arquivos internos e externos os elementos de informação que possam ser representados por classes do modelo conceitual, em orientação a objetos, ou tabelas, no modelo relacional, por exemplo. Arquivos internos são aqueles mantidos pela própria aplicação, enquanto que arquivos externos são aqueles usados pela aplicação, mas mantidos externamente.

O que vai determinar se um arquivo é interno ou externo então é a definição da *borda do sistema*, ou seja, os limites daquilo que é considerado interno ao sistema sendo desenvolvido ou analisado e de outros sistemas. No caso de aplicações cliente/servidor, deve-se considerar

[94] www.softwaremetrics.com/Function%20Point%20Training%20Booklet%20New.pdf

que a borda do sistema passa por ambos os módulos, porque nenhum deles isoladamente se constitui em uma aplicação completa com significado para o usuário.

Aqui convém fazer uma ressalva, já que o método APF original diferencia os *Record Element Types* (*RET*), que correspondem a um subconjunto de dados reconhecível pelo usuário dentro de um arquivo interno ou externo e os *File Types Referenced* (*FTR*), que podem conter os RET. Fazendo-se um paralelo com orientação a objetos, os RET seriam classes quaisquer, enquanto que os FTR seriam classes que não são componentes nem agregados de outras classes. Por exemplo, se um CD é composto por músicas, então apenas o CD poderia ser um FTR, mas tanto o CD quanto a música poderiam ser RET.

Assim, um FTR é uma classe que, caso participe de uma composição ou agregação, ocupa o lugar mais alto da hierarquia. Um FTR pode *ter* componentes, mas não pode *ser* componente.

Essa distinção é importante, porque o cálculo da complexidade das transações (entrada, saída e consulta) é baseado em FTR, enquanto que o cálculo da complexidade dos arquivos internos e externos é baseado em RET. Resumindo, a complexidade das transações contabiliza apenas as classes no mais alto nível de uma hierarquia de composição, enquanto que a complexidade de arquivos contabiliza quaisquer classes, inclusive as componentes.

As funções identificadas nos requisitos devem, então, ser classificadas de acordo com a sua natureza:

a) *Entradas externas*. São entradas de dados ou controle, que tem como consequência a alteração do estado interno das informações do sistema.
b) *Saídas externas*. São saídas de dados que podem ser precedidas ou não da entrada de parâmetros. Pelo menos um dos dados de saída deve ser derivado, ou seja, calculado.
c) *Consultas externas*. São saídas de dados que podem ser precedidas ou não da entrada de parâmetros. Os dados devem sair da mesma forma como estavam armazenados, sem transformações ou cálculos.
d) *Arquivo interno*. É um elemento do modelo conceitual percebido pelo usuário e mantido internamente pelo sistema.
e) *Arquivo externo*. É um elemento do modelo conceitual percebido pelo usuário e mantido externamente por outras aplicações.

Entradas externas são, então, funções que pegam dados ou controle do usuário ou de outras aplicações e levam para dentro do sistema. A função deve ter como objetivo armazenar, alterar ou remover dados de forma direta (alterando os dados no sistema) ou indireta (solicitando a outra aplicação que faça a alteração nos dados).

Ou seja, dados passados pelo usuário com o único fim de servir como *parâmetros* para uma consulta ou saída não devem ser considerados como entradas.

Por outro lado, um comando que passa o identificador de um registro (por exemplo, o CPF de um cliente) para deletar um conjunto de dados (por exemplo, deletar o cliente) pode ser considerado como entrada, pois é um controle que remove dados, ou seja, faz uma mudança no estado interno das informações do sistema.

Então, entradas são funções que adicionam, alteram ou deletam informação no sistema. Seus argumentos são os dados passados como parâmetro para localizar os objetos a serem alterados e os parâmetros que correspondem aos novos valores, quando for o csao.

Algumas vezes, as operações de alteração e exclusão ocorrem em dois passos, ou seja, são precedidas de uma consulta. Esta consulta será contabilizada à parte, como consulta externa. Por exemplo, um usuário que deseja fazer a alteração do cadastro de um cliente, primeiramente entra com o CPF do cliente e visualiza na tela seus dados (isso é uma consulta externa), em seguida, ele altera os dados que deseja e salva as novas informações (isso é uma entrada externa).

Além disso, se uma entrada externa puder produzir um conjunto de mensagens de erro, por conta de possíveis exceções, cada mensagem de erro vai contar como um argumento da entrada (conforme será visto adiante). Essas mensagens não devem ser contabilizadas como saídas independentes.

Saídas externas são funções que pegam dados de dentro do sistema e apresentam ao cliente ou enviam a outras aplicações em sua forma original ou transformada, sendo que, pelo menos um valor derivado (calculado) deve existir para que seja uma saída externa (caso contrário é consulta externa).

As saídas, então, são os fluxos de informação para fora do sistema. Não se conta como saída externa uma função que simplesmente pega dados do sistema e apresenta esses dados da forma como estão (isso será contabilizado como uma *consulta* externa).

Mas se houver qualquer tipo de operação matemática ou lógica sobre esses dados, então a função pode ser considerada como uma saída.

Uma saída externa também pode ter parâmetros. Por exemplo, passa-se o CPF de um cliente e uma data, e a partir desses parâmetros se obtém um relatório de todas as vendas realizadas para este cliente a partir da data definida, com seu total. Como a data e o CPF são usados para filtrar as vendas que devem aparecer no relatório e fazer a totalização, essa função deve ser considerada como saída externa e não como uma mera consulta.

A *consulta externa* consiste da apresentação de dados exatamente da mesma forma como foram armazenados, sem cálculos ou transformações. A consulta normalmente inclui parâmetros de entrada, como por exemplo, passar o CPF de uma pessoa para obter seus dados de cadastro (isso é uma simples consulta).

Um *arquivo interno* é uma informação complexa (uma classe, em orientação a objetos) do tipo FTR, ou seja, uma classe que não é componente de outras classes, embora possa ter seus próprios componentes.

Um *arquivo externo* é uma informação complexa (uma classe, em orientação a objetos) do tipo FTR, que é mantida em outras aplicações, ou seja, não é gerenciada pela aplicação que se vai desenvolver.

Como os arquivos internos e externos devem ser do tipo FTR, então, por exemplo, se o modelo conceitual possui uma classe CD que agrega a classe Música, apenas a classe CD será considerada arquivo interno ou externo, pois Música faz parte de CD (por agregação ou composição), não sendo considerada um arquivo interno ou externo à parte.

A maioria dos arquivos internos e externos tem associadas transações de consulta, entrada e saída, que são contabilizadas à parte.

Nota-se que quando da verificação da complexidade dessas funções, apenas a complexidade visível pelo usuário será considerada. A complexidade algorítmica interna não é aplicada às funções individuais (até porque a princípio ela ainda pode não ser conhecida), mas estimada para o projeto como um todo, da mesma forma que os fatores multiplicadores de esforço de CII (Seção 7.3.2).

7.4.2 UFP - Pontos de Função não Ajustados

Uma vez determinado o tipo da função (transação ou arquivo), sua complexidade vai ser calculada a partir dos seguintes fatores:

a) *Registro* (*RET - Record Element Type*), que corresponde a um subconjunto de dados reconhecível pelo usuário dentro de um arquivo interno ou externo (uma classe qualquer).
b) *Arquivo* (*FTR - File Types Referenced*), que corresponde a um arquivo interno ou externo, usado em uma transação (uma classe que não seja componente de outra).
c) *Argumento* (*DET - Data Element Type*), que corresponde a uma unidade de informação (um campo), a princípio indivisível e reconhecível pelo usuário, normalmente seria um campo de uma tabela, um atributo de uma classe ou um parâmetro de uma função.

A complexidade das transações é determinada pela quantidade de arquivos FTR e argumentos DET. Já a complexidade dos arquivos internos e externos é determinada pela quantidade de registros RET e argumentos DET.

No caso das entradas externas, os DET podem ser os campos de entrada de informação, mensagens de erro e botões que podem ser pressionados, por exemplo.

No caso das saídas externas, os DET podem ser os campos em um relatório, valores calculados, mensagens de erro e cabeçalhos de colunas que são lidos de um arquivo interno.

No caso das consultas externas os DET podem ser os campos usados para pesquisa e os campos mostrados como resposta à consulta. Se houver vários botões para fazer a consulta de maneira diferente, cada um deles deve contar como um DET também.

Nota-se que as saídas e consultas podem ter tanto dados de entrada (parâmetros) como de saída (resultados). As entradas externas normalmente só têm campos de entrada (a informação que se vai armazenar), mas as mensagens de erro potencialmente produzíveis por uma entrada podem ser também contabilizadas como um dado DET.

A complexidade de uma transação é, portanto, calculada a partir da quantidade de FTR e DET. O FTR corresponde ao número de classes (que não são componentes ou agregados de outra classe) do modelo conceitual que contêm as informações de entrada e/ou saída como

atributos. Por exemplo, se uma função de entrada passa uma data que será armazenada como atributo da classe X e um valor numérico que será armazenado como atributo de uma classe Y, então existem duas classes envolvidas e o FTR neste caso é igual a 2. Se os dois valores vão ser armazenados como atributos de uma mesma classe, ou como atributos de dois componentes de uma mesma classe, ou ainda como atributos de uma classe e uma de suas componentes, então considera-se que há apenas um FTR. Por exemplo, uma função que grava o nome de um CD e o nome das músicas deste CD tem apenas um FTR (o CD), mesmo que as músicas sejam consideradas classes à parte, como uma música faz parte do CD ela não pode ser contabilizada como FTR.

O segundo valor usado para calcular a complexidade de uma transação é o seu número de argumentos, ou DET, ou seja, a quantidade de valores de entrada e/ou saída, independentemente de quais classes eles pertença. Assim, por exemplo, uma função de entrada que envia 5 valores ao sistema terá 5 argumentos, e o DET será igual a 5. Uma consulta que passa 2 parâmetros e recebe 8 valores como retorno, tem 10 argumentos DET.

Deve-se ainda saber que o que conta, no caso de DET é o *tipo* de valor. Caso a função envie ou receba uma lista de valores de um mesmo tipo, a lista conta uma única vez. Por exemplo, uma função que passa um nome de pessoa e uma lista de telefones, terá 2 argumentos (mesmo que essa lista tenha 10 telefones, ela conta uma única vez). Por outro lado, se algum tipo de cálculo for feito com a lista de valores, então cada cálculo contará como um argumento diferente. Por exemplo, uma saída que apresente uma lista de notas, a média e desvio padrão dessas notas, deverá ter considerados 3 argumentos DET.

As funções de entrada têm sua complexidade calculada de acordo com a Tabela 7-38.

Tabela 7-38: Complexidade funcional de entradas externas.

	Argumentos DET		
Classes FTR	**1 a 4**	**5 a 15**	**16 ou mais**
0 a 1	Baixa	Baixa	Média
2	Baixa	Média	Alta
3 ou mais	Média	Alta	Alta

As saídas e consultas têm sua complexidade calculada de acordo com a Tabela 7-39.

Tabela 7-39: Complexidade funcional de saídas e consultas.

	Argumentos DET		
Classes FTR	**1 a 5**	**6 a 19**	**20 ou mais**
0 a 1	Baixa	Baixa	Média
2 a 3	Baixa	Média	Alta
4 ou mais	Média	Alta	Alta

Já os arquivos internos e externos têm sua complexidade calculada em função de classes quaisquer (inclusive componentes de outras classes), ou seja, RET. Além de RET deve-se usar DET como no caso de transações para determinar a complexidade dos arquivos. Deve ser então aplicada a Tabela 7-40.

Tabela 7-40: Complexidade funcional de arquivos internos e externos.

	Argumentos DET		
Classes RET	**1 a 19**	**20 a 50**	**51 ou mais**
1	Baixa	Baixa	Média
2 a 5	Baixa	Média	Alta
6 ou mais	Média	Alta	Alta

Embora a maioria dos arquivos do tipo CRUD relacione-se com uma única classe (exemplo, cadastro de Clientes), pode haver informações que, por possuírem subcomponentes, acabam se relacionando com mais de uma classe, apesar de ainda poderem ser consideradas como um cadastro único. É o caso, por exemplo, do padrão Mestre-Detalhe, onde duas classes são tratadas juntas como um único cadastro, como: livro e capítulo, CD e música, passagem e trecho, etc. Assim, um arquivo interno ou externo será a classe de mais alto nível na hierarquia de composição (FTR) e o número de classes RET da Tabela 7-40 será o número de componentes ou subcomponentes dela mais um. Por exemplo, CD poderia ser um arquivo interno (FTR) com duas classes RET: CD e Música.

Depois de determinar a complexidade de cada transação e arquivo, pode-se obter o seu número de pontos de função não ajustados UFP aplicando a Tabela 7-41.

Tabela 7-41: Pontos de função não ajustados por tipo e complexidade de função.

	Complexidade funcional		
Tipo de função	**Baixa**	**Média**	**Alta**
Entrada	3	4	6
Saída	4	5	7
Consulta	3	4	6
Arquivo interno	7	10	15
Arquivo externo	5	7	10

O número UFP de pontos de função não ajustados para o sistema como um todo será simplesmente a soma dos pontos de função não ajustados obtidos para cada uma das funções do sistema.

A identificação das transações e arquivos de um sistema é o ponto chave para determinar sua complexidade. Porém, muitas vezes requisitos são incompletos ou muito genéricos para que uma boa contagem possa ser feita.

Mas a própria técnica de pontos de função pode ajudar a melhorar a qualidade dos requisitos. Em primeiro lugar, a técnica necessita dos argumentos DAT; então, será necessário revisar se os requisitos efetivamente listam todas as informações elementares necessárias para realizar cada função.

Em segundo lugar, a identificação de arquivos e transações caminham juntas. Uma vez identificado um arquivo interno ou externo (classe), pode-se imaginar que pelo menos as transações mais básicas para esse arquivo devam existir. Essas transações, usualmente chamadas CRUD ou CRUDL, implicam na criação (entrada externa), alteração (entrada

externa), remoção (entrada externa), consulta (consulta externa) e listagem (consulta externa). Nem sempre todas estão presentes na aplicação. É possível por exemplo que a aplicação crie um arquivo de dados que só vai ser consultado por outra aplicação. Mas sempre é de se pensar se elas existem ou não como transações.

Porém, nem sempre um arquivo sofre apenas essas transações tão básicas em seu ciclo de vida. Um livro, por exemplo, em uma livraria virtual, não será apenas criado, alterado, consultado e deletado. Em determinados momentos ele será colocado em oferta, reservado ou vendido. Essas transações podem ser consideradas como simples alterações, já incluídas no CRUDL?

A técnica responde a essa questão da seguinte forma: se uma transação de alteração de um objeto, como "colocar em oferta" implica em usar apenas o conjunto ou um subconjunto de DET, e os arquivos FTR ou um subconjunto deles, já considerados na operação de alteração do CRUDL, então, a alteração já é contemplada pelas transações CRUDL. Por exemplo, se colocar um livro em oferta implica em simplesmente mudar um valor de um atributo dele, o que já pode ser feito pela operação CRUDL de alteração, então "colocar um livro em oferta" não será uma nova função e não vai contar pontos de função.

Por outro lado, se a alteração implica em usar dados DET não incluídos no conjunto da operação de alteração ou ainda arquivos FTR não incluídos na alteração CRUDL, então a transação deverá ser considerada uma nova função e contar seus próprios pontos de função. Por exemplo, se "colocar livro em oferta", implica em obter dados de um arquivo de ofertas padrão, o que não é feito quando da operação de alteração CRUDL, então trata-se de uma nova função.

A título de exemplo de aplicação da técnica, considere-se a seguinte lista de requisitos:

a) O sistema deve permitir o gerenciamento (CRUDL) de informações sobre livros e usuários. Dos livros inclui-se: título, ISBN, autor, número de páginas, editora e ano de publicação. Dos usuários inclui-se: nome, documento, endereço, telefone e email.
b) O sistema deve permitir o registro de empréstimos onde são informados o documento do usuário e o ISBN de cada um dos livros.
c) Quando um empréstimo for executado, o sistema deve armazenar as informações em uma tabela relacional usando chaves estrangeiras para identificar o usuário e os livros.
d) Após o registro de um empréstimo deve ser impresso um recibo com o nome do usuário, e título e data de devolução prevista para cada livro que deve ser calculada como a data atual somada ao prazo do livro.

Analisando-se o primeiro requisito, percebe-se que se trata de dois cadastros independentes com suas operações CRUDL. O segundo requisito é uma entrada externa. O terceiro requisito é uma função interna que não deve ser contabilizada. O quarto requisito será considerado uma saída externa, pois realiza um cálculo para obter a data de entrega em função da data atual e do prazo do livro. As funções a serem consideradas então, são apresentadas na Tabela 7-42.

Tabela 7-42: Exemplo de identificação de funções a partir de requisitos.

Função	Tipo	FTR	RET	DET	#FTR	#RET	#DET	Complex.	UFP
Cadastro de Livros	Arquivo interno		Livro, Editora, Autor	título, isbn, autor, número de páginas, editora, ano de publicação		3	6	Baixa	7
• Inserir Livro	Entrada externa	Livro, Editora, Autor		título, isbn, autor, número de páginas, editora, ano de publicação	3		6	Alta	6
• Alterar Livro	Entrada externa	Livro, Editora, Autor		título, isbn, autor, número de páginas, editora, ano de publicação	3		6	Alta	6
• Excluir Livro	Entrada externa	Livro		título, isbn, ano de publicação	1		3	Baixa	3
• Consultar Livro	Consulta externa	Livro, Editora, Autor		título, isbn, autor, número de páginas, editora, ano de publicação	3		6	Média	4
• Listar Livros	Consulta externa	Livro, Editora, Autor		título, isbn, autor, número de páginas, editora, ano de publicação	3		6	Média	4
Cadastro de Usuários	Arquivo interno		Pessoa	nome, documento, endereço, telefone, email		1	5	Baixa	7
• Inserir Usuário	Entrada externa	Pessoa		nome, documento, endereço, telefone, email	1		5	Baixa	3
• Alterar Usuário	Entrada externa	Pessoa		nome, documento, endereço, telefone, email	1		5	Baixa	3
• Excluir Usuário	Entrada externa	Pessoa		nome, documento	1		2	Baixa	3
• Consultar Usuário	Consulta externa	Pessoa		nome, documento, endereço, telefone, email	1		5	Baixa	3
• Listar Usuários	Consulta externa	Pessoa		nome, documento	1		2	Baixa	3
Registrar empréstimo	Entrada externa	Pessoa, Livro, Emprestimo		nome da pessoa, isbn dos livros	3		2	Média	4
Imprimir recibo	Consulta externa	Pessoa, Livro, Emprestimo		nome da pessoa, título do livro, data de hoje, prazo do livro, data de devolução	3		5	Baixa	3
TOTAL									**59**

Portanto, aplicando-se a técnica às funções identificadas nos requisitos acima, considerando que o modelo conceitual tenha as classes Pessoa, Livro, Emprestimo, Editora e Autor, sem nenhuma relação de agregação ou composição entre elas, chega-se a um cálculo de 59 pontos de função não ajustados. Note-se que se Editora e Autor não fossem considerados como

classes (arquivos), mas meramente como campos a serem adicionados à classe livro, a complexidade das funções relacionadas com Livro seria menor, pois haveria menos classes envolvidas.

Por outro lado, a consideração dessas informações como classes leva a conclusão de que os requisitos ainda não estão completos, pois Editora e Autor deveriam ser consideradas como cadastros e, portanto, como arquivos internos ou externos com suas respectivas operações CRUDL. Assim, após essa revisão nos requisitos o número de pontos de função não ajustado deveria ser recalculado.

7.4.3 AFP – Pontos de Função Ajustados

O método de pontos de função não tem fatores de escala como COCOMO. Então ele presume que o esforço será linear em relação à quantidade de funcionalidades implementadas. Porém, o método possui um conjunto de fatores de ajuste técnico já que diferentes projetos e diferentes equipes poderão produzir funcionalidades em ritmos diferentes. Por exemplo, um simples *front-end* para banco de dados terá suas funções desenvolvidas muito mais rapidamente do que um software para controle de um sistema de distribuição de energia elétrica. Assim, a funcionalidade dada por UFP, ou pontos de função não ajustados, não leva em consideração a complexidade técnica interna das funções, mas apenas a percepção que um usuário tem delas.

Para se chegar a valores de esforço de desenvolvimento mais realistas, então, será necessário ajustar esse valor a partir dos fatores técnicos.

A técnica de pontos de função sugere 14 fatores de ajuste técnico, conhecidos como GSC (*General Systems Characteristics*). Todos têm o mesmo peso, e a eles deve ser atribuída uma nota de zero a cinco, onde zero significa que o fator não tem nenhuma influência no projeto, e cinco significa que o fator tem influência determinante do projeto, sendo os valores de 1 a 4 intermediários.

Os 14 GSC são os seguintes (mais detalhes na Seção 7.4.5):

a) Comunicação de dados.
b) Processamento de dados distribuído.
c) Performance.
d) Uso do sistema.
e) Taxa de transações.
f) Entrada de dados *online*.
g) Eficiência do usuário final.
h) Atualização *online*.
i) Processamento complexo.
j) Reusabilidade.
k) Facilidade de instalação.
l) Facilidade de operação.
m) Múltiplos locais.
n) Facilidade para mudança.

A avaliação dos GSC é feita para o projeto como um todo (não para cada função). Então, como são 14 GSC e cada um receberá uma nota de 0 a 5, a somatória total das notas ficará entre 0 e 70. Esse valor ajustado é conhecido como *VAF* (*Value Adjustment Factor*):

$$VAF = 0{,}65 + \left(0{,}01 * \sum_{i=1}^{14} GSC_i\right)$$

Assim, o somatório dos *GSC* multiplicado por 0,01 e somado a 0,65 vai fazer com que *VAF* varie de 0,65 a 1,35.

Esse valor é multiplicado pelo número de UFP para obter o AFP, ou número de pontos de função ajustados:

$$AFP = UFP * VAF$$

Assim, em um projeto onde todos os fatores técnicos sejam mínimos (nota 0), o *AFP* será igual a 65% do valor nominal de *UFP*. Já em um sistema onde todos os fatores técnicos sejam máximos (nota 5) o *AFP* será igual a 135% do valor nominal de *UFP*. Um sistema nominal seria aquele onde todos os fatores técnicos tenham nota 3, o que levaria o *AFP* a ser igual ao *UFP*.

7.4.4 Duração e Custo de um Projeto

Uma vez que o *AFP* do projeto tenha sido calculado, o esforço total será calculado multiplicando-se o *AFP* pelo índice de produtividade (*IP*) da equipe. Esse índice deve ser calculado para o ambiente local, e pode variar muito em função do ambiente de trabalho, experiência da equipe e outros fatores. Assim, o esforço total do projeto é calculado como:

$E = AFP * IP$

O *IP* de uma equipe pode ser calculado da seguinte forma: toma-se um projeto já desenvolvido, para o qual se saiba quanto esforço foi despendido. O esforço *E* deve ser contado em desenvolvedor-mês, ou seja, se 4 pessoas trabalharam durante 10 meses, então o valor do esforço é de 40 desenvolvedor-mês. Para este mesmo projeto, estima-se os pontos de função ajustados a partir dos requisitos iniciais (a versão dos requisitos que existia antes de iniciar o desenvolvimento propriamente dito). Se, por exemplo, eram 800 AFP, então o índice de produtividade da equipe para este projeto é: $IP = AFP/E = 800/40 = 20$. Neste caso, o índice de produtividade da equipe é de 20 pontos de função ajustados por desenvolvedor-mês.

Se houver mais de um projeto disponível para cálculo pode-se somar os AFP de todos os projetos e dividir pelo esforço despendido em todos os projetos, desta forma obtendo uma média aritmética deles.

Já o custo do projeto é calculado como o esforço total multiplicado pelo custo médio da hora do desenvolvedor e ambiente:

$Custo = E * Custo_{hora}$

Há um *site*[95] na Internet que armazena editais brasileiros de contratação de software onde a medida de custo é o ponto de função. No site, o preço por ponto de função varia de 100 a 1000 reais, a maioria ficando entre 400 e 600 reais, o que pode ser explicado pelo tipo de sistema que se está contratando.

O tempo linear e tamanho médio da equipe podem ser calculados como em COCOMO (Seção 7.2) ou CII (Seção 7.3).

Outra maneira mais simples para calcular o tempo linear ideal, sem usar KSLOC, é através da seguinte equação:

$$T = 2{,}5 * \sqrt[3]{E}$$

Para esta equação funcionar é necessário que o esforço *E* seja expresso em *desenvolvedor-mês*. O uso de outras unidades como desenvolvedor-semana ou desenvolvedor-hora vai provocar distorções no resultado.

O tamanho médio da equipe, neste caso será:

$$P = E/T$$

Além do tempo linear ideal, pode-se também falar em tempo mínimo de projeto, caso exista urgência para desenvolver o projeto e se possa gastar mais recursos para tentar desenvolve-lo num prazo mais curto. O tempo mínimo de um projeto pode ser calculado da seguinte forma:

$$T_{min} = 0{,}75 * \sqrt[3]{E}$$

Neste caso, é bem possível que o tamanho médio da equipe deva ser maior do que E/T_{min}. Deve-se também ter em mente que um projeto nestas condições estará sendo gerenciado em condições limite, e, portanto, apenas equipes muito experientes e organizadas, a princípio, teriam capacidade para dar conta deste ritmo de desenvolvimento.

Todavia, essas fórmulas são, de certa forma, "mágicas", devido à grande variedade de projetos e equipes de desenvolvimento de software. Antes de usá-las seriamente em uma empresa convém testá-las e ajustá-las sempre para a realidade local.

7.4.5 Detalhamento dos Fatores Técnicos

Nesta seção será detalhada a interpretação dos fatores técnicos que compõem o VAF (Longstreet, 2012)[96], para que notas consistentes possam ser atribuídas.

O GSC *comunicação de dados* avalia o grau em que necessidades especiais de comunicação afetam o sistema. Sistemas isolados estariam no extremo inferior de avaliação, já os sistemas que fazem uso intensivo de dados obtidos em outros lugares e que enviam informação para muitos lugares diferentes, através de diferentes protocolos de comunicação, estariam no extremo oposto. As notas são atribuídas assim:

[95] www.fattocs.com.br/editais.asp

[96] www.softwaremetrics.com/Function%20Point%20Training%20Booklet%20New.pdf

a) 0: para aplicações que são somente processamento em *batch* ou que rodam isoladas em um PC.
b) 1: para aplicações em *batch* mas com entrada de dados remota *ou* saída remota.
c) 2: para aplicações em *batch* com entrada de dados remota *e* saída remota.
d) 3: para aplicações que incluem coleta de dados *online* ou *front-end* de teleprocessamento para um sistema em *batch* ou sistema de consultas.
e) 4: para aplicações que são mais do que um *front-end*, mas suportam um único tipo de protocolo de comunicação.
f) 5: para aplicações que são mais do que um *front-ent* e suportam vários tipos de protocolos de comunicação.

O fator *processamento de dados distribuído* avalia o grau em que dados distribuídos são usados pela aplicação. No extremo inferior estão os sistemas que armazenam os dados todos no mesmo lugar localmente, ou que, tendo necessidade de transferir dados, nada fazem para automatizar essa transferência. No extremo superior estão as aplicações que distribuem o processamento dinamicamente entre diferentes nodos, escolhendo sempre o melhor nodo possível para efetuar o processamento específico. As notas são atribuídas assim:

a) 0: para aplicações que não ajudam na transferência de dados ou funções de processamento entre os componentes do sistema.
b) 1: para aplicações que preparam os dados para o processamento do usuário final em outro componente do sistema, tal como sistemas que geram dados para serem lidos em uma planilha ou arquivo de processador de texto.
c) 2: para aplicações que preparam dados para transferência e então transferem e processam os dados em outro componente do sistema (não para processamento do usuário final).
d) 3: para aplicações onde o processamento distribuído e transferência de dados ocorrem *online* e em uma direção apenas.
e) 4: para aplicações onde o processamento distribuído e transferência de dados ocorrem *online* e nas duas direções.
f) 5: para aplicações onde as funções são executadas dinamicamente no componente mais apropriado do sistema.

O fator *performance* avalia o grau em que a eficiência do sistema precisa ser considerada em sua construção. Sistemas eficientes sempre são desejáveis, mas este fator avalia o quanto a eficiência é critica para o sistema, de forma que se invistam recursos de tempo e dinheiro para melhorar esse aspecto. Um sistema de empréstimo de vídeos em uma videolocadora poderia estar no extremo inferior, porque normalmente trabalha com quantidades de dados pequenas e algoritmos simples, de forma que a eficiência não será uma preocupação. No extremo superior poder-se-ia ter sistemas que trabalham com quantidades absurdamente grandes de dados e/ou precisam apresentar respostas rápidas, como por exemplo, um sistema que interpreta dados de geoprocessamento em tempo real, ou um sistema que controla a injeção eletrônica de combustível em um automóvel. As notas são atribuídas assim:

a) 0: nenhum requisito de performance especial foi definido pelo cliente.

b) 1: requisitos de performance foram estabelecidos e revisados, mas nenhuma ação especial precisa ser tomada.
c) 2: tempo de resposta e taxa de transferência são críticos durante as horas de pico. Nenhum *design* especial para utilização de CPU é necessário. O prazo para a maioria dos processamentos é o dia seguinte.
d) 3: o tempo de resposta e taxa de transferência são críticos durante o horário comercial. Nenhum *design* especial para utilização de CPU é necessário. Os requisitos de prazo de processamento com sistemas interfaceados são restritivos.
e) 4: em adição, os requisitos de performance são suficientemente restritivos para que se necessite estabelecer tarefas de análise de performance durante a fase de *design*.
f) 5: em adição, ferramentas de análise de performance devem ser usadas nas fases de *design*, desenvolvimento e/ou implementação para atender os requisitos de performance do cliente.

O fator *uso do sistema* avalia o grau em que o sistema necessita ser projetado para compartilhar recursos de processamento. No extremo inferior estão as aplicações para as quais não há nenhuma preocupação com compartilhamento de hardware, e no extremo superior as aplicações para as quais o hardware estará sendo compartilhado enquanto existem restrições operacionais da aplicação em si. As notas são atribuídas assim:

a) 0: nenhuma restrição operacional implícita ou explícita é incluída.
b) 1: restrições operacionais existem mas são menos restritivas do que em uma aplicação típica. Nenhum esforço especial é necessário para satisfazer as restrições.
c) 2: são incluídas algumas considerações sobre tempo e segurança.
d) 3: requisitos específicos de processador para uma parte específica da aplicação são incluídos.
e) 4: restrições sobre operações estabelecidas requerem que a aplicação tenha um processador dedicado ou prioridade de tempo no processador central.
f) 5: em adição, existem restrições especiais na aplicação em relação aos componentes distribuídos do sistema.

O fator *taxa de transações* avalia a quantidade de transações simultâneas esperada. Os limites inferior e superior podem variar em função do desenvolvimento tecnológico do hardware e das redes de comunicação. O que há 10 anos poderia ter sido considerado difícil em termos de taxa de transações, hoje pode ser trivial. Então, é necessário que se avalie, em termos da tecnologia atual, qual seria a taxa de influência deste fator no desenvolvimento do produto. No limite inferior estariam sistemas que teriam tão poucas transações que a tecnologia atual daria conta delas sem nenhuma preocupação extra. No extremo superior estariam sistemas no limite da tecnologia, os quais poderiam precisar de vários containers de processadores servidores para poder atender aos usuários (em 2012, na casa de possíveis milhões de acessos simultâneos). Exemplos de sistemas com essa taxa de acesso são o Facebook®[97] e o Google®[98]. As notas são atribuídas assim:

a) 0: não são antecipados períodos de picos de transações.

[97] www.facebook.com
[98] www.google.com

b) 1: períodos de picos de transações (por exemplo, mensalmente, semestralmente, anualmente) são antecipados.
c) 2: picos de transação semanais são antecipados.
d) 3: picos de transação diários são antecipados.
e) 4: altas taxas de transação são estabelecidas pelo cliente nos requisitos da aplicação ou nos acordos de nível de serviço, as quais são suficientemente altas para necessitar de atividades de análise de performance na fase de *design*.
f) 5: em adição, se requer o uso de ferramentas de análise de performance nas fases de *design*, desenvolvimento e/ou instalação.

O fator *entrada de dados online* avalia a percentagem de informação que o sistema deve obter *online*, ou seja, dos usuários em tempo real. No extremo inferior estão os sistemas que tomam toda a informação necessária de arquivos ou repositórios. No extremo superior os sistemas nos quais mais de 30% das transações são entradas de dados *online*. As notas são atribuídas assim:

a) 0: todas as transações são processadas em modo *batch*.
b) 1: 1% a 7% das transações são entradas de dados interativas.
c) 2: 8% a 15% das transações são entradas de dados interativas.
d) 3: 16% a 23% das transações são entradas de dados interativas.
e) 4: 24% a 30% das transações são entradas de dados interativas.
f) 5: mais de 30% das transações são entradas de dados interativas.

O fator *eficiência do usuário final* avalia o grau em que a aplicação será projetada para melhorar a eficiência do usuário final. Não ter preocupação nenhuma com isso leva o sistema ao limite inferior deste fator. Uma preocupação extrema em melhorar a eficiência do trabalho do usuário leva ao limite superior. As notas são atribuídas a partir de uma lista de itens que podem ser considerados ou não:

a) Ajuda navegacional (por exemplo, teclas de função, menus gerados dinamicamente, etc.).
b) Menus.
c) Ajuda e documentação *online*.
d) Movimentação de cursor automatizada.
e) *Scrolling*.
f) Impressão remota (a partir de transações *online*).
g) Teclas de função pré-definidas.
h) Tarefas em *batch* submetidas a partir de transações *online*.
i) Seleção por cursor na tela de dados.
j) Alto uso de cores e destaque visual em tela.
k) Copias impressas de documentação de usuário de transações *online*.
l) Interface por mouse.
m) Janelas *pop-up*.
n) Minimização do número de janelas para realizar objetivos de negócio.
o) Suporte bilíngue (conta como quatro itens).
p) Suporte multilíngue (conta como seis itens).

As notas são então calculadas em função da lista acima.

a) 0: nenhuma das opções acima.
b) 1: de uma a três das opções acima.
c) 2: de quatro a cinco das opções acima.
d) 3: seis ou mais das opções acima, mas não há requisitos específicos relacionados a eficiência de usuário final.
e) 4: seis ou mais das opções acima, e requisitos estabelecidos para a eficiência de usuário final são suficientemente fortes para requerer a inclusão de atividades de *design* para fatores humanos (por exemplo minimizar a quantidade de *clicks* e movimentos de mouse, maximização de *defaults* e uso de *templates*).
f) 5: seis ou mais das opções acima, e os requisitos estabelecidos para a eficiência de usuário são suficientemente fortes para requerer o uso de ferramentas e processos especiais para demonstrar que os objetivos foram atingidos.

O fator *atualização online* avalia o percentual de arquivos internos que podem ser atualizados de forma *online*. Se nenhum arquivo interno pode ser atualizado *online* então deve-se atribuir a nota mínima a esse fator. Quando todos os principais arquivos podem ser atualizados *online*, e existem restrições especiais de volume e segurança, então se está no extremo superior do fator. As notas são atribuídas assim:

a) 0: nenhuma atualização *online*.
b) 1: é incluída a atualização *online* para um a três arquivos. O volume de atualização é baixo e a recuperação é simples.
c) 2: a atualização *online* de quatro ou mais arquivos é incluída. O volume de atualização é baixo e a recuperação é simples.
d) 3: a atualização *online* dos principais arquivos lógicos internos é incluída.
e) 4: em adição, proteção contra a perda de dados é essencial, e o sistema deve ser especialmente projetado contra perda de dados.
f) 5: em adição, altos volumes de atualização trazem considerações de custo para o processo de recuperação. Procedimentos de recuperação altamente automatizados com intervenção mínima do operador são incluídos.

O fator *processamento complexo* avalia o grau em que a aplicação utiliza processamento lógico ou matemático complexo. No extremo inferior estariam sistemas que apenas armazenam dados e eventualmente fazem algum tipo de soma ou produto, e no outro extremo sistemas baseados em inteligência artificial e processamento numérico complexo na área de física e engenharia.

Os seguintes componentes são considerados para avaliação da complexidade do processamento da aplicação:

a) Controle cuidadoso (por exemplo, processamento especial de auditoria) e/ou processamento seguro específico da aplicação.
b) Processamento lógico extensivo.
c) Processamento matemático extensivo.

d) Muito processamento de exceções resultante de transações incompletas que precisam ser processadas novamente, como, por exemplo, transações de caixa-automático incompletas causadas por interrupção de teleprocessamento, valores de dados que faltam ou edições que falharam.
e) Processamento complexo para gerenciar múltiplas possibilidades de entrada e saída, como, por exemplo, multimídia ou independência de dispositivos.

As notas são então atribuídas em função destes itens:

a) 0: nenhuma das opções acima.
b) 1: qualquer uma das opções acima.
c) 2: quaisquer duas das opções acima.
d) 3: quaisquer três das opções acima.
e) 4: quaisquer quatro das opções acima.
f) 5: todas as cinco opções acima.

O fator *reusabilidade* avalia em que grau a aplicação é projetada para ser reusável. No extremo inferior estão aplicações que são desenvolvidas sem nenhuma preocupação com reusabilidade e no extremo superior aplicações desenvolvidas para gerar vários componentes parametrizados reusáveis. As notas são atribuídas assim:

a) 0: não há nenhuma preocupação para produzir código reusável.
b) 1: código reusável é gerado para uso dentro da própria aplicação.
c) 2: menos de 10% da aplicação deve considerar mais do que simplesmente as necessidades do usuário.
d) 3: 10% ou mais da aplicação deve considerar mais do que as necessidades do usuário.
e) 4: a aplicação deve ser especificamente empacotada e/ou documentada para facilitar o reuso, e a aplicação deve ser personalizável pelo usuário em nível de código fonte.
f) 5: a aplicação deve ser especificamente empacotada e/ou documentada para facilitar o reuso, e a aplicação deve ser personalizável por meio de manutenção de usuário baseada em parâmetros.

O fator *facilidade de instalação* avalia em que grau haverá preocupação em tornar fácil a instalação do sistema e a conversão dos dados. Um sistema para o qual não haja necessidade de conversão de dados e que deva ser instalado sem nenhuma automatização estará no limite inferior. Um sistema que deva se auto-instalar automaticamente e converter automaticamente dados de sistemas legados estará no limite superior. As notas são atribuídas assim:

a) 0: nenhuma consideração especial foi estabelecida pelo usuário, e nenhum *setup* especial é necessário para a instalação.
b) 1: nenhuma consideração especial foi estabelecida pelo usuário, mas um *setup* especial é requerido para instalação.
c) 2: requisitos de conversão e instalação foram estabelecidos pelo usuário, e guias de conversão e instalação devem ser fornecidas e testadas. O impacto da conversão no projeto não é considerado importante.

d) 3: requisitos de conversão e instalação de foram estabelecidos pelo usuário, e guias de conversão e instalação devem ser fornecidas e testadas. O impacto da conversão no projeto é considerado importante.
e) 4: em adição à nota 2 acima, ferramentas de conversão e instalação automática devem ser fornecidas e testadas.
f) 5: em adição à nota 3 acima, ferramentas de conversão e instalação automática devem ser fornecidas e testadas.

O fator *facilidade de operação* avalia em que grau a aplicação deverá fazer automaticamente processos de *startup*, recuperação de falhas e *backup*. Aplicações que não façam nada destas coisas automaticamente estão no limite inferior, e aplicações que automatizem totalmente os processos de operação estão no limite superior. As notas são atribuídas assim:

a) 0: nenhuma consideração operacional especial além dos procedimentos normais de *backup* foram estabelecidos pelo usuário.
b) 1-4: um, alguns ou todos os itens abaixo se aplicam ao sistema (Deve-se selecionar todos os que se aplicam. Cada item vale um ponto, exceto se for dito o contrário):
    a. Processos efetivos de inicialização, *backup* e recuperação devem ser fornecidos, mas a intervenção do operador é necessária.
    b. Processos efetivos de inicialização, *backup* e recuperação devem ser fornecidos, e nenhuma intervenção do operador é necessária (conta como dois itens).
    c. A aplicação deve minimizar a necessidade de armazenamento em fitas (ou qualquer outro meio de armazenamento *offline*).
    d. A aplicação deve minimizar a necessidade de manuseio de papel.
c) 5: a aplicação é projetada para operar de forma não supervisionada. Não supervisionada significa que não é necessária nenhuma intervenção do operador do sistema a não ser, talvez, na sua primeira inicialização ou desligamento final. Uma das características da aplicação é a recuperação automática de erros.

O fator *múltiplos locais* avalia o grau em que a aplicação é projetada para funcionar de forma distribuída. No limite inferior estão as aplicações isoladas *stand-alone* e no superior as aplicações formadas por diferentes módulos que rodam em máquinas geograficamente distribuídas ou dispositivos móveis. As notas são atribuídas assim:

a) 0: requisitos do usuário não exigem a consideração de necessidade de mais do que um usuário ou instalação.
b) 1: a necessidade de múltiplos locais deve ser considerada no projeto, e a aplicação deve ser projetada para operar apenas em ambientes idênticos de hardware e software.
c) 2: a necessidade de múltiplos locais deve ser considerada no projeto, e aplicação deve ser projetada para operar apenas em ambientes de hardware e software similares.
d) 3: a necessidade de múltiplos locais deve ser considerada no projeto e aplicação é projetada para operar em ambientes de hardware e software diferentes.
e) 4: o plano de documentação e suporte deve ser fornecido e testado para suportar a aplicação em múltiplos locais, e a aplicação é como descrita nas notas um ou dois.

f) 5: o plano de documentação e suporte deve ser fornecido e testado para suportar a aplicação em múltiplos locais, e a aplicação é como descrita na nota três.

O fator *facilidade para mudança* avalia o grau em que a aplicação é projetada para facilitar mudanças lógicas e estruturais. Aplicações desenvolvidas sem esse tipo de preocupação estão no limite inferior, e aplicações projetadas para acomodar extensivamente mudanças futuras estarão no limite superior.

As seguintes características podem se aplicar ao software:

a) Facilidades de consulta e relatório flexíveis devem ser fornecidas para tratar consultas simples, por exemplo, operadores lógicos binários aplicados apenas a um arquivo lógico interno (conta como um item).
b) Facilidades de consulta e relatório flexíveis devem ser fornecidas para tratar consultas de complexidade média, por exemplo, operadores lógicos binários aplicados a mais do que um arquivo lógico interno (conta com dois itens).
c) Facilidades de consulta e relatório flexíveis devem ser fornecidas para tratar consultas de complexidade alta, por exemplo, combinações de operadores lógicos binários em um ou mais arquivos lógicos internos (conta como três itens).
d) Dados de controle de negócio são mantidos em tabelas que são gerenciadas pelo usuário e com processos interativos *online*, mas as mudanças só têm efeito no dia seguinte (contra como um item).
e) Dados de controle de negócio são mantidos em tabelas que são gerenciadas pelo usuário e com processos interativos *online*, e as mudanças têm feito imediatamente (conta como dois itens).

Considerando os itens acima, a nota deve então ser estabelecida assim:

a) 0: nenhum item.
b) 1: um item.
c) 2: dois itens.
d) 3: três itens.
e) 4: quatro itens.
f) 5: cinco itens ou mais.

7.5 Pontos de Caso de Uso

A técnica de *Pontos de Caso de Uso* surgiu em 1993 a partir da Tese de Gustav Karner (1993). O método é baseado em Análise de Pontos de Função, especificamente MK II, que é um modelo relativamente mais simples que o do IFPUG.

A técnica foi incorporada como parte do Processo Unificado, já que este é fortemente baseado em casos de uso. Um dos maiores problemas para a aplicação do método não está no método em si, mas na falta de padronização de entendimento sobre o que é efetivamente um caso de uso (Anda, 2001). Uma explicação detalhada sobre este assunto é apresentada nos capítulos 4 e 5 de Wazlawick (2011). Em resumo, um caso de uso deve ser:

a) Um processo de negócio que tenha significado para o usuário.

b) Um processo completo, com início e fim, que deixa a informação do sistema em um estado consistente.
c) Um processo iterativo onde informações entram e saem do sistema para os atores.
d) Um processo que é executado em uma única sessão de uso do sistema, ou seja, que não pode ser interrompido. Uma vez iniciado, ou o caso de uso vai até seu final (ou um de seus finais) ou é abortado.

O método se baseia na análise da quantidade e complexidade dos atores e casos de uso, o que gera os UUCP, ou pontos de caso de uso não ajustados. Depois, a aplicação e fatores técnicos e ambientais leva aos UCP, ou pontos de caso de uso (ajustados).

7.5.1 UAW - Complexidade de Atores

Os atores são pessoas ou sistemas externos à aplicação em análise, com os quais a aplicação se comunica. O método de classificação de complexidade de atores é bem simples. Cada ator é contado uma única vez, mesmo que apareça em vários casos de uso. No caso de atores que são especializações de um ator mais genérico, conta-se apenas as especializações mais elementares, ou seja, aqueles atores que não possuem subtipos.

Uma vez identificados os atores, sua complexidade é definida da seguinte forma:

a) *Atores humanos que interagem com o sistema através de interface gráfica* são considerados complexos e recebem 3 pontos de caso de uso.
b) *Sistemas que interagem por um protocolo como TCP/IP* e *atores humanos que interagem com o sistema apenas por linha de comando* são considerados de média complexidade, e recebem 2 pontos de caso de uso.
c) *Sistemas que são acessados por interfaces de programação* (*API*) são considerados de baixa complexidade, e recebem 1 ponto de caso de uso.

O valor de *UAW* (*Unajusted Actor Weight*) é, então, simplesmente, a soma dos pontos atribuídos a todos os atores relacionados no sistema.

O uso da medida UAW na estimação de esforço não é, porém, uma unanimidade. Há autores que recomendam que apenas os casos de uso sejam ponderados, e que os atores sejam ignorados na contagem. Inclusive ferramentas de contagem usualmente colocam o uso de UAW como uma medida opcional, cujo uso fica a critério do analista.

7.5.2 UUCW – Complexidade dos Casos de Uso

Os casos de uso também devem ser contados uma única vez. Apenas processos completos devem ser contados, ou seja, extensões, variantes e casos de uso incluídos não devem ser contados.

Na proposta original de Karner, a complexidade de um caso de uso era definida em função do número estimado de transações (movimentos de informação para dentro ou para fora do sistema), incluindo as sequências alternativas do caso de uso. Assim, o casos de uso seriam assim classificados:

a) Casos de uso *simples* devem possuir no máximo 3 transações, e recebem 5 pontos de caso de uso.

b) Casos de uso *médios* devem possuir de 4 a 7 transações, e recebem 10 pontos de caso de uso.
c) Casos de uso *complexos* devem possuir mais de 7 transações, e recebem 15 pontos de caso de uso.

Uma forma alternativa de estimar a complexidade de um caso de uso é em função da quantidade de classes necessária para implementar as funções do caso de uso. Assim:

a) Casos de uso simples devem ser implementados com 5 classes ou menos.
b) Casos de uso médios devem ser implementados com 6 a 10 classes.
c) Casos de uso complexos devem ser implementados com mais de 10 classes.

Outra forma ainda de estimar a complexidade de um caso de uso é pela análise de seu risco. Assim:

a) Casos de uso como relatórios, têm apenas uma ou duas transações e baixo risco, pois não alteram dados, e podem ser considerados casos de uso simples.
b) Casos de uso padronizados, como CRUD, têm um número conhecido e limitado de transações, têm médio risco (pois embora a lógica de funcionamento seja conhecida, regras de negócio obscuras podem existir), e podem ser considerados como casos de uso médios.
c) Casos de uso não padronizados têm um número desconhecido de transações e alto risco, pois além das regras de negócio serem desconhecidas, ainda deve-se descobrir qual é o fluxo principal e quais as sequências alternativas. Assim, esse tipo de caso de uso deverá ser considerado como complexo.

O valor de *UUCW* (*Unajusted Use Case Weight*) é dado pela soma dos valores atribuídos a cada um dos casos de uso da aplicação.

7.5.3 UUCP – Pontos de Caso de Uso não Ajustados

O valor de pontos de caso de uso não ajustados, ou UUCP, é calculado simplesmente como:

$$UUCP = UAW + UUCW$$

Como mencionado anteriormente, o peso dos atores pode ser opcionalmente excluído do cálculo, o que faria com que *UUCP* fosse imediatamente igual a *UUCW*.

7.5.4 TCF - Fatores Técnicos

Pontos de caso de uso fazem o ajuste dos pontos em função de dois critérios: fatores técnicos (que pertencem ao projeto) e fatores ambientais (que pertencem à equipe).

Cada fator recebe uma nota de 0 a 5, onde 0 indica nenhuma influência do projeto, 3 é a influência nominal e 5 máxima influência no projeto.

Os fatores técnicos são 13, e, ao contrário do método de pontos de função, cada um tem um peso específico, conforme mostrado na Tabela 7-43.

Tabela 7-43: Fatores técnicos de ajuste de pontos de caso de uso.

Sigla	Fator	Peso
T1	Sistema Distribuído	2
T2	Performance	2
T3	Eficiência de usuário final	1
T4	Complexidade de processamento	1
T5	Projeto visando código reusável	1
T6	Facilidade de instalação	0,5
T7	Facilidade de uso	0,5
T8	Portabilidade	2
T9	Facilidade de mudança	1
T10	Concorrência	1
T11	Segurança	1
T12	Acesso fornecido a terceiros	1
T13	Necessidades de treinamento	1

Então, cada um dos 13 fatores terá uma nota de 0 a 5 multiplicada pelo seu peso. A somatória desses produtos corresponde ao *TFactor*, ou impacto dos fatores técnicos, cujo valor varia entre 0 e 75.

O valor *TCF* (*Technical Complexity Factor*) pode então ser calculado fazendo-se a normalização do *TFactor* para o intervalo de 0,6 a 1,35:

$$TCF = 0{,}6 + (0{,}01 * TFactor)$$

7.5.5 EF – Fatores Ambientais

Um aspecto que distingue a técnica de pontos de caso de uso de pontos de função e CII é que ela tem um fator de ajuste específico para as características da equipe de desenvolvimento. Assim, pode-se tomar o mesmo projeto, com os mesmos fatores técnicos, e ele poderá ter pontos de caso de uso ajustados diferentes para equipes diferentes.

Os fatores ambientais são oito, e avaliam o ambiente de trabalho. Cada um também tem um peso diferente e, inclusive, dois deles tem peso negativo. Novamente as notas devem variar de 0 a 5, mas o significado das notas é um pouco diferente das atribuídas aos fatores técnicos. Enquanto as notas dos fatores técnicos medem a influência dos fatores no projeto, as notas dos fatores ambientais medem a qualidade dos fatores no ambiente de trabalho. Por exemplo, o fator “motivação”, com nota zero significa que a equipe não está nem um pouco motivada, com nota 3 significa que a motivação da equipe é média, ou normal, já a nota 5 significa que a equipe está altamente motivada.

Assim, os fatores ambientais são definidos conforme a Tabela 7-44.

Tabela 7-44: Fatores ambientais de ajuste de pontos de caso de uso.

Sigla	Fator	Peso
E1	Familiaridade com o processo de desenvolvimento	1,5
E2	Experiência com a aplicação	0,5
E3	Experiência com orientação a objetos	1
E4	Capacidade do analista líder	0,5
E5	Motivação	1
E6	Estabilidade de requisitos obtida historicamente	2
E7	Equipe em tempo parcial	-1
E8	Dificuldade com a linguagem de programação	-1

Assim, o valor das notas multiplicado pelo peso dos fatores ambientais pode variar de -10 até 32,5. Esse valor é chamado de *EFactor*.

O *EFactor* é normalizado para o intervalo de 0,425 a 1,7. Assim, os fatores ambientais são calculados da seguinte forma:

$$EF = 1{,}4 - (0{,}03 * EFactor)$$

7.5.6 UCP – Pontos de Caso de Uso Ajustados

Uma vez calculados os pesos dos atores e casos de uso, os fatores técnicos e ambientais, os pontos de caso de uso ajustados podem ser obtidos pela simples multiplicação destes três valores:

$$UCP = UUCP * TCF * EF$$

7.5.7 Esforço

Uma vez calculado o número de pontos de caso de uso ajustados, isso deve ser transformado em esforço pela aplicação do índice de produtividade da equipe em termos de pontos de caso de uso:

$$E = UCP * IP$$

Assim como no caso de pontos de função, o *IP* pode ser calculado tomando-se projetos já desenvolvidos, somando-se o tempo trabalhado por cada desenvolvedor e dividindo pelo número de pontos de caso de uso estimados. Assim, se o esforço real de um conjunto de projetos foi de 900 desenvolvedor-hora e o *UCP* total destes projetos era de 60, então o índice de produtividade dessa equipe é de 900/60, ou seja 15 horas de trabalho por desenvolvedor por ponto de caso de uso.

A literatura de pontos de caso de uso refere-se normalmente a *horas por UCP*, enquanto outros métodos como COCOMO preferem o uso da medida de *desenvolvedor-mês*. Considerando que um mês equivale a 158 horas de trabalho, neste caso, tem-se que 15 horas por UCP equivalem a cerca de 10,5 (=158/15) *UCP* por desenvolvedor-mês.

O trabalho original de Karner (1993) estimava que 20 horas por ponto de caso de uso ajustado seria um bom valor médio. Posteriormente, Ribu (2001) determinou que esse valor poderia variar de 15 a 30.

Outro trabalho (Schneider & Winters, 1998) propõem que os fatores ambientais E1 a E6 com nota menor do que 3 sejam somados à quantidade de fatores ambientais E7 a E8 com nota maior do que 3:

a) Se o total for 2 ou menos, assumir 20 horas por UCP.
b) Se o total for 3 ou 4, assumir 28 horas por UCP.
c) Se o total for maior do que 4, os fatores ambientais apresentam um risco muito grande ao projeto e devem ser melhorados antes que se assuma qualquer compromisso de desenvolvimento, ou então, assumir 36 horas por UCP.

7.5.8 Métodos de Contagem para Casos de Uso Detalhados

A técnica de contagem de pontos de caso de uso apresentada até o momento considera apenas uma *estimativa* de complexidade de casos de uso porque ela trabalha com casos de uso de alto nível, os quais são representados unicamente pelo seu nome ou eventualmente uma ou duas frases explicativas adicionais.

Existem técnicas de contagem de casos de uso detalhados, mas sua utilidade é mais limitada, pois no Processo Unificado os casos de uso só serão detalhados normalmente durante os ciclos iterativos. Na fase de planejamento de projeto, normalmente, os analistas vão contar apenas com casos de uso de alto nível.

Mesmo assim, essas técnicas podem ser úteis, se os casos de uso estiverem detalhados, ou caso se queira refinar as estimativas de esforço no início de um ciclo iterativo. Por este motivo elas serão resumidas aqui.

Robiolo e Orosco (2008) sugerem que o texto do caso de uso detalhado seja analisado e que sejam contadas as entidades e transações. Dessa forma o tamanho de uma aplicação em pontos de caso de uso não ajustados seria a somatória destes valores.

Braz e Vergilio (2006) propõem uma métrica chamada *FUSP* (*Fuzzy Use Case Size Points*) cuja contagem é baseada na complexidade dos atores que participam do caso de uso, na complexidade das pré-condições e pós-condições do caso de uso, no somatório do número de entidades referenciadas e passos do caso de uso e na complexidade dos fluxos alternativos do caso de uso, sejam variantes ou tratadores de exceção. Assim, a medida é bastante criteriosa, mas bastante trabalhosa, e precisa ter os casos de uso totalmente detalhados para poder ser aplicada.

Mohagheghi, Anda e Conradi (2005), propõem uma técnica (*Adapted Use Case Points*) que assume que inicialmente todos os atores são de complexidade média e todos os casos de uso de complexidade alta. Posteriormente, os casos de uso devem ser quebrados em casos de uso menores e reclassificados em simples e médios, se for o caso, a medida que o processo de refinamento vai ocorrendo. Casos de uso estendidos e fluxos alternativos também são contabilizados.

Kamal e Ahmed (2011)[99] apresentam uma extensiva comparação entre estes e outros métodos de contagem de pontos de caso de uso.

7.5.9 FUCS – *Full Use Case Size*

O método *FUCS*, ou *Full Use Case Size* (Ibarra, 2011) foi criado com o objetivo de medir o tamanho de casos e uso para gerar uma medida de complexidade de software. O método mede aspectos funcionais e não funcionais e, para isso, necessita como entrada, os casos de uso de alto nível e a lista de requisitos do sistema. A medição de um caso de uso é baseada em três componentes:

a) O tipo de caso de uso.
b) O número de operações de sistema e regras de negócio.
c) O número de requisitos de interface.

O método identifica, primeiramente, dez tipos de casos de uso:

a) *Operação básica sobre uma entidade.* É um caso de uso que realiza apenas uma das operações básicas CRUD, ou seja, inserir, alterar, excluir ou consultar um objeto. Esse tipo de caso de uso corresponde à menor unidade funcional possível com significado para o usuário.
b) *Consulta parametrizada.* É um tipo de caso de uso onde as informações sobre um ou mais objetos são recuperadas a partir de um ou mais parâmetros. O caso de uso normalmente tem dois passos: (1) o ator passa parâmetros e (2) o sistema envia dados. Esse tipo de caso de uso pode apresentar inúmeras variantes, como consultar cliente por data, consultar cliente por produto comprado, consultar cliente por bairro; mas com a utilização de técnicas como *objeto-filtro* (Wazlawick, 2011, p. 223), todas essas variações podem ser implementadas em uma única função.
c) *Relatório.* É semelhante à consulta parametrizada, mas ao invés de retornar apenas os dados armazenados dos objetos, retorna dados derivados (totalizações, por exemplo) e organiza esses dados em grupos e seções, por filtros e ordenação
d) *CRUD.* Corresponde à união das quatro operações básicas sobre uma entidade, e, sempre que possível deve ser usado ao invés daquelas.
e) *CRUD tabular.* Corresponde a um CRUD onde um grupo de objetos da mesma classe pode ser editado ao mesmo tempo, ou seja, os dados são apresentados como uma tabela onde possivelmente as linhas contêm os objetos e as colunas seus atributos. Normalmente é usado apenas no caso de cadastros mais simples, com poucos atributos e poucos objetos, pois, do contrário, a utilização da tabela deixa de ser prática.
f) *CRUD mestre/detalhe.* É uma variação do CRUD onde existem pelo menos duas classes com uma relação de agregação ou composição entre elas (podem existir mais níveis de agregação ou composição). No caso do mestre/detalhe com duas classes, existe a classe Mestre, que representa o "todo" e a classe detalhe, que representa as partes do todo, como por exemplo, um livro e seus capítulos ou um CD e suas músicas.

[99] www.sdiwc.net/noahjohn/web-admin/upload-pdf/00000078.pdf

g) *CRUD detalhes.* É uma variação do CRUD mestre/detalhe onde os dados do mestre podem eventualmente ser apenas visualizados (ou nem isso), mas não alterados. Assim, apenas os atributos da classe detalhe poderão ser gerenciados pelo usuário.
h) *Serviço.* São casos de uso que implementam a interação automática com outros sistemas, na maioria das vezes sem intervenção de um usuário humano. Esses casos de uso normalmente não necessitam de interfaces gráficas. Um exemplo seria um caso de uso de sincronização diária de arquivos de uma aplicação com outro sistema.
i) *Biblioteca de função ou componente.* Esse tipo de caso de uso não é propriamente um processo de usuário com início e fim, mas um fragmento de processo. Por vezes, o modelador de casos de uso se defronta com funcionalidades que são repetidas várias vezes em outros casos de uso, como, por exemplo, "receber pagamento", mas que sozinhas não são um processo completo (deve-se pagar por algum produto ou serviço). Mas, por serem reusáveis, esses fragmentos de caso de uso acabam sendo representados no diagrama como extensões ou como fragmentos incluídos em outros casos de uso. Dependendo da situação, até mesmo uma biblioteca de funcionalidades de caso de uso pode ser criada para ser reusada ao longo do projeto ou até em outros projetos.
j) *Não padronizado.* Trata-se dos casos de uso complexos, não classificáveis pelos critérios acima, e cuja diversidade é tão grande que não justifica a criação de padrões, já que eles raramente se repetem.

A atribuição de pontos aos tipos de caso de uso é definida em função de três perguntas:

a) Existe um modelo de especificação disponível para o caso de uso?
b) Existe algum modelo de interface que pode ser usado para o caso de uso?
c) Existe algum *framework* de suporte para este tipo de caso de uso? Em média, qual o percentual de redução de esforço obtido com o uso do *framework*.

O método sugere que todos os casos de uso iniciem com cinco pontos e, em seguida, tenham desconto em função das respostas às três perguntas acima da seguinte forma:

a) Se a resposta à questão (a) for "sim", então subtraia um ponto.
b) Se a resposta à questão (b) for "sim", então subtraia (mais) um ponto.
c) Se a resposta à questão (c) for "sim", então se o percentual de redução for de 20 a 40%, subtraia (mais) um ponto, se for superior a 40%, subtraia (mais) dois pontos.

Conforme o tipo de caso de uso, e a existência de modelos e tecnologia, então, uma pontuação específica UC_{TP} entre um e cinco será atribuída a cada tipo. Ibarra (2011, p. 107) apresenta uma possível avaliação, conforme a Tabela 7-45.

Tabela 7-45: Atribuição de pontos aos casos de uso de acordo com seu tipo.

Tipo	UC_{TP}
Operação básica sobre uma entidade	1
CRUD	2
CRUD Tabular	2
CRUD Mestre/Detalhe	4
CRUD Detalhes	3
Consulta Parametrizada	1
Relatório	1
Serviço	5
Biblioteca de Função ou Componente	5
Não Padronizado	5

Essa pontuação, porém, pode variar, a medida que novos padrões e *frameworks* forem definidos para os respectivos tipos de caso de uso. Além disso, novos tipos podem ser identificados e sua pontuação definida de acordo com o método.

A essa pontuação serão acrescidos mais dois componentes:

a) Número de operações de sistema extras e regras de negócio.
b) Número de requisitos de interface com usuário.

Operações de sistema devem ser acrescidas ao caso de uso sempre que ele apresente alguma transação (entrada ou saída de informação) extra, em relação ao seu padrão definido. Por exemplo, um caso de uso CRUD que inclui uma operação de geração de senha para o usuário (uma função que não pertence ao padrão CRUD) deve ter seu valor em pontos de caso de uso aumentado, pois sua implementação custará mais esforço.

Da mesma forma, regras de negócio aumentam a complexidade de um caso de uso. Regras de negócio são usualmente requisitos não funcionais não tecnológicos, ou seja, restrições sobre como as funções do sistema são executadas. Um exemplo seria: “a totalização das comissões deve ser feita sempre no 5º dia útil e referir-se aos pedidos faturados no mês anterior que já tenham sido pagos”.

Assim, para cada caso de uso, a técnica propõe que sejam contadas as operações de sistema extras e as regras de negócio e que seu valor seja somado ao valor do caso de uso. A influência destes fatores, porém, ainda vai depender do grau de esforço α que tais elementos produzem, o que pode ser avaliado da seguinte forma:

a) α = 5 quando não existem ferramentas, padrões e *frameworks* que facilitem a implementação de operações de sistema e regras de negócio.
b) α = 3 quando existe um conjunto de bibliotecas padrão com pontos de extensão que facilitam a implementação deste tipo de requisito.
c) α = 1 quando além das bibliotecas existem ferramentas que possibilitam implementar esses requisitos de forma visual ou automatizada.

Além disso, os requisitos de interface gráfica costumam ter um grande peso no desenvolvimento de um sistema, pois a interface gráfica é o que o usuário vê e usa. Assim, o

método também propõe que estes requisitos sejam contados e somados ao valor do caso de uso. Um exemplo de requisito de interface é "quando o usuário digita o CEP o sistema deve preencher automaticamente o endereço". O peso relativo β dos requisitos de interface será definido a partir das seguintes questões:

a) β = 5 quando não existem ferramentas, padrões e *frameworks* que facilitem a implementação de recursos de interface gráfica.
b) α = 3 quando existe um conjunto de bibliotecas padrão e componentes de interface gráfica que facilitam a implementação deste tipo de requisito.
c) α = 1 quando, além das bibliotecas e componentes de interface gráfica, existem ferramentas que possibilitam implementar esses requisitos de forma visual ou automatizada.

Assim, o tamanho de cada caso de uso é dado pela fórmula:

$$UC_S = UC_{TP} + \alpha * N_{SOBR} + \beta * N_{GUIR}$$

Onde:

a) UC_S é o tamanho (*use case size*) do caso de uso.
b) UC_{TP} é o tamanho do caso de uso em função de seu tipo (Tabela 7-45).
c) α é o peso das operações de sistema e regras de negócio, conforme definido acima.
d) N_{SOBR} é o número total de operações de sistema extras e regras de negócio do caso de uso.
e) β é o peso dos requisitos de interface com usuário.
f) N_{GUIR} é o número total de requisitos de interface com usuário do caso de uso.

Um dos principais diferenciais deste método em relação aos outros é que ele não classifica os casos de uso em categorias de complexidades. Os casos de uso são medidos em UC_S (que varia de 1 a infinito) em função de seu tipo e do número de operações de sistema, regras de negócio e requisitos de interface com o usuário. O tamanho do sistema será simplesmente a somatória do tamanho de cada um de seus casos de uso. A transformação desse valor em esforço depende somente de multiplicar o valor pelo índice de produtividade da equipe.

7.6 Pontos de Histórias

Pontos de História[100] (*PH*) é a estimativa de esforço preferida (embora não exclusiva) de métodos ágeis como *Scrum* e XP. Um ponto de história, não é uma medida de complexidade funcional como pontos de função ou pontos de caso de uso, mas uma medida de esforço relativa à equipe de desenvolvimento.

Segundo Kniberg (2007) uma estimativa baseada em pontos de histórias deve ser feita pela equipe. Inicialmente pergunta-se à equipe quanto tempo tantas pessoas que se dedicassem unicamente a uma história de usuário levariam para terminá-la, gerando uma versão executável funcional. Se a resposta for, por exemplo, "3 pessoas levariam 4 dias", então atribua à história 3x4 = 12 pontos de história.

[100] Story Points

Assim, um ponto de história pode ser definido como o esforço de desenvolvimento de uma pessoa durante um dia ideal de trabalho, lembrando que o *dia ideal* de trabalho consiste em uma pessoa dedicada durante 6 a 8 horas a um projeto, sem interrupções nem atividades paralelas.

7.6.1 Atribuição de Pontos de História

Nos métodos ágeis, a importância da estimativa normalmente está na comparação entre histórias, ou seja, mais importante do que saber quantos dias uma história efetivamente levaria para ser implementada é saber que uma história levaria duas vezes mais tempo do que outra para ser implementada. Por este motivo, os pontos de história são atribuídos normalmente não como valores da série dos números naturais, mas como valores da série aproximada de números de Fibonacci[101]. Um número de Fibonacci é definido como a soma dos dois números de Fibonacci anteriores na série (com exceção dos dois primeiros, que por definição são 1 e 1). Assim, o início da série de Fibonacci é constituído pelos números: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, etc. Porém, pode ser estranho mensurar pontos de história em 89 ou 34 pontos. Então na prática acaba-se fazendo uma aproximação desses valores para uma série como: 1, 2, 3, 5, 8, 15, 25, 40, 60, 100, etc. A ideia é que os pontos de história apresentem uma ordem de grandeza natural para o esforço e não uma medida exata.

Outra opção para estimar pontos de história é usar o sistema "camiseta", com valores "pequeno", "médio" e "grande".

O procedimento de atribuição de pontos funciona assim: toma-se da lista de histórias de usuário previamente preparada aquelas consideradas mais simples, e atribui-se a elas 1 ou 2 pontos. Depois, sequencialmente, vai-se pegando outras mais complexas, inicialmente de 3, depois de 5 pontos e assim por diante. Segundo Toledo (2009)[102], o motivo é que, para o ser humano é muito mais fácil fazer medidas relativas do que absolutas. É difícil uma pessoa estimar o peso de um cavalo, sem ter uma balança ou conhecimento prévio do valor. Mas uma pessoa consegue estimar facilmente que um cavalo pesa menos do que um elefante e mais do que um cachorro.

Felix (2009)[103] comenta que a atribuição de pontos de história usualmente segue critérios subjetivos de complexidade, esforço e risco, sendo caracterizada por frases como, por exemplo:

a) *Complexidade*: "Essa regra de negócio tem muitos cenários possíveis".
b) *Esforço*: "Essa alteração é simples, mas precisa ser realizada em muitas telas".
c) *Risco*: "Precisamos utilizar o *framework* X, mas ninguém na equipe tem experiência".

O esforço, nos métodos ágeis, precisa ser estimado pela equipe como um todo, e não por um gerente, porque justamente é medida a capacidade das pessoas em realizar as tarefas. Então é necessário que as pessoas que vão efetivamente trabalhar nestas tarefas possam opinar e avaliar a carga de trabalho, que, depois de decidida, passa a ser um contrato de desenvolvimento, ou seja, um compromisso.

[101] pt.wikipedia.org/wiki/N%C3%BAmero_de_Fibonacci
[102] visaoagil.files.wordpress.com/2009/01/storypoints.pdf
[103] lucianofelix.wordpress.com/2009/06/10/o-que-significam-story-points/

7.6.2 Medição de Velocidade

Pontos de histórias são usados por equipes ágeis para medir sua velocidade de projeto. Pode-se fazer um gráfico e deixar a vista de todos onde a cada ciclo são medidos os pontos de história efetivamente desenvolvidos. Se esta velocidade começar a cair, a equipe deve verificar o motivo. Vários motivos podem ser listados: desmotivação, erros de estimação, erros de priorização (a equipe começou a tratar histórias de usuário mais simples e deixou as mais complexas e arriscadas para depois), etc.

Em todo o caso, quando se deseja aumentar a velocidade da equipe é comum que se faça a aquisição de mais desenvolvedores para trabalhar nos ciclos restantes. É natural que enquanto os novos desenvolvedores estejam se ambientando, a velocidade em PH seja reduzida, para mais tarde aumentar. A*** Figura 7-4 mostra um exemplo de gráfico de velocidade de projeto onde após a aquisição de novos membros para a equipe há uma redução na velocidade em PH que é compensada após 3 ciclos, quando então os novos membros da equipe passam a ser efetivamente produtivos.

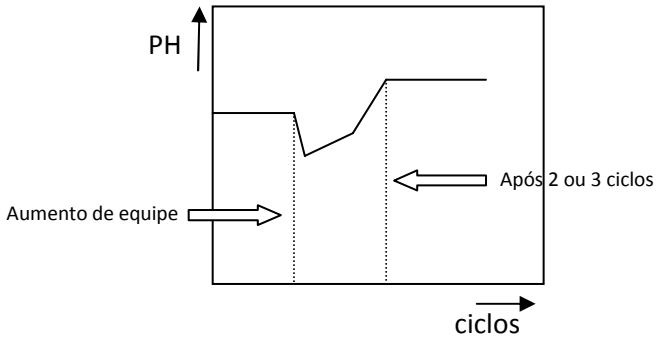

Figura 7-4: Gráfico de velocidade de projeto em pontos de história[104].

Normalmente um gráfico de velocidade é relativamente estável, embora os valores possam variar de um ciclo para outro, sua derivada se mantém constante. Se houver medidas de melhoria de produtividade pode-se esperar aumentos na velocidade, ou seja, uma derivada positiva. Então, a principal utilidade do gráfico de velocidade consiste em ajudar a diagnosticar possíveis problemas de ambiente, caso a derivada se torne negativa.

[104] Fonte: Toledo (2009).

