2
Tecnologias Envolvidas

Neste capítulo serão apresentadas as tecnologias mais importantes envolvidas neste trabalho, que são: software embarcado, software de monitoramento e aquisição de dados, sistemas de tempo real, sistemas orientados a recuperação, agentes de software, *design by contract*, *bluetooth*, biblioteca log4cxx, biblioteca SQLite, biblioteca Qt, *mock objects* e componentes de software.

Apesar de estarem senda apresentadas de forma isolada, é importante ressaltar que algumas destas tecnologias estão intimamente relacionadas. Este relacionamento pode ser notado ao longo do texto, pois ao descrever uma tecnologia, muitas vezes é necessário recorrer a outra.

2.1
Software Embarcado

Sistemas de software embarcados[15] são caracterizados pela capacidade de executar em ambientes isolados, realizando tarefas muito bem definidas e, muitas vezes, com nenhuma ou pouca interação com usuários. Tais sistemas são encontrados com freqüência no dia-a-dia, estando presentes em equipamentos que vão desde máquinas industriais e eletrodomésticos, a celulares e PDAs.

Os ambientes de execução de sistemas de software embarcados geralmente possuem limitações de tempo e de hardware, alguns não possuindo discos rígidos, sistema operacional, teclado ou monitor. Tais sistemas também são conhecidos como *firmware* ou *embedded software*.

Muitos sistemas embarcados são projetados para serem capazes de tomar decisões por si só baseados no estado do ambiente no qual executam. Muitas vezes, tais sistemas possuem severos requisitos de confiabilidade e de tolerância a falhas, pois uma falha de execução oriunda da má programação pode acarretar prejuízos inaceitáveis, muitas vezes envolvendo não só perdas financeiras, mas também colocando vidas em risco.

Outra classe de sistemas embarcados é a classe que envolve severos requisitos em relação à performance no tempo de execução. Tais sistemas são chamados de sistemas de tempo real, onde pode-se identificar outro grande grupo de possíveis falhas, referentes à desobediência nas especificações do tempo de resposta do sistema.

Sistemas de software embarcados também são muito utilizados para monitoração, controle e aquisição de dados.

2.2 Software de Monitoramento e Aquisição de Dados

Software de monitoramento e aquisição são responsáveis por monitorar o estado ou o comportamento de um sistema ou dispositivo. Por exemplo: um sistema responsável por monitorar sensores de temperatura com o objetivo de controlar a temperatura de uma caldeira; um sistema de segurança responsável por monitorar sensores de movimento a fim de verificar a existência de elementos se movendo (em primeira aproximação, pessoas) em um ambiente; etc.

Normalmente software desse gênero são responsáveis por sistemas que devem ser a prova de falhas, pois suas conseqüências podem incorrer em perdas financeiras inaceitáveis ou até mesmo em perda de vidas - por exemplo, sistemas que monitoram a temperatura de reatores em usinas nucleares. Estes software normalmente são embarcados e estão integrados ao sistema ou dispositivo que monitoram.

2.3 Sistemas de Tempo Real

Sistemas de tempo real precisam ser capazes de responder a estímulos dentro de um limite de tempo (tipicamente milissegundos ou microssegundos).[16] Um sistema que monitore a temperatura de um motor e tenha que ser capaz de desligá-lo rapidamente caso a temperatura exceda um certo limite é um exemplo de sistema de tempo real.

Sistemas de tempo real[15] podem ser classificados como rígidos (*hard*) ou leves (*soft*). É possível diferenciar um sistema de tempo real rígido de um leve pelo fato de que o primeiro pode causar perdas irreparáveis (e, por isso, inaceitáveis) caso não sejam rigorosamente satisfeitos os seus requisitos de tempo de resposta.

Sistemas de tempo real rígidos e embarcados geralmente interagem em baixo nível com o hardware físico, fazendo com que seja necessário projetar o

sistema como um todo, envolvendo hardware e software e, em seguida, alocar as parcelas de controle ao software. Dessa forma obtém-se, dentre outras, as restrições de desempenho e capacidade a serem satisfeitas. Um exemplo de sistema de tempo real rígido e embarcado é o sistema de controle do air-bag de um carro, que pode ocasionar mortes caso não seja acionado na rapidez especificada.

Os sistemas leves são aqueles que continuam funcionando mesmo que seus requisitos de tempo de resposta não sejam satisfeitos. Nestes sistemas quando o tempo de resposta não é respeitado, ocorre uma execução deficiente (com atraso, queda de qualidade, etc.), mas não uma interrupção do serviço. Aplicações de áudio e vídeo ao vivo são típicos exemplos de onde a violação do tempo causa perda da qualidade do serviço, mas não inviabiliza a operação do sistema.

2.4 Sistemas Orientados à Recuperação

Segundo Patterson[5] os sistemas orientados à recuperação devem ser construídos com base no fato de que falhas de hardware ou software e erros de operação do sistema são fatos com os quais é preciso conviver, e não problemas que possam ser adequadamente resolvidos e completamente eliminados durante o desenvolvimento do software.

Um software orientado à recuperação deve ter seu desenvolvimento focado nos seguintes objetivos:

- Minimizar o risco de existência de anomalias presentes no software.
- Minimizar a suscetibilidade do software a eventos danosos externos.
- Reduzir o tempo médio de reparação (TMRP[1]) do software.
- Reduzir o tempo médio de recuperação (TMRC[2]) do software.
- Minimizar as conseqüências das falhas.

[1]TMRP é o tempo decorrido desde o momento da identificação da falha até o momento em que a(s) falta(s) responsável(is) é (são) consertada(s).

[2]TMRC é o tempo decorrido desde o momento em que uma falha interrompe um serviço até o momento em que esse serviço é restabelecido, ainda que de forma degradada, mas asseguradamente fidedigna.

2.4.1
Como Construir um Sistema Orientado à Recuperação

A partir de sua própria definição, pode-se dizer que o foco de um sistema orientado à recuperação é a capacidade de co-existir com faltas. Dessa forma, o esforço de desenvolvimento deve ser direcionado para esse fim. Há quatro pontos importantes a serem abordados na concentração de esforços quando se deseja construir um sistema orientado à recuperação que são:[17]

Esforço na prevenção de faltas: É o esforço despendido em tempo de desenvolvimento com o objetivo de evitar o risco de presença de faltas em um sistema. Este esforço se concentra nas fases de especificação, projeto, codificação, teste e homologação de um sistema.

Esforço na detecção de falhas: É o esforço despendido em tempo de desenvolvimento com o objetivo de detectar falhas em tempo de execução do sistema. É refletido em artefatos como: hardware dedicado; validadores de estruturas de dados; algoritmos de auto-teste; e o uso de redundância de software ou hardware a fim de detectar inconsistências por comparação de diferentes saídas resultantes de uma mesma entrada.

Esforço no tratamento de falhas: É o esforço despendido em tempo de desenvolvimento para o tratamento de falhas[3] detectadas em tempo de execução. Por exemplo, uma falha detectada pelo software de controle de um determinado sensor pode, ao invés de encerrar a execução, resultar em uma execução onde o sensor defeituoso é ignorado (o sistema opera como se tivesse um sensor a menos). Em outros casos, a melhor recuperação possível (a fim de minimizar a perda de dados) é encerrar o programa. A recuperação pode requerer intervenção do usuário para fornecer informações que minimizem a perda de dados. O usuário também pode solicitar que a recuperação não seja efetuada, caso deseje continuar a execução com o sistema instável e, neste caso, o sistema deve avaliar se a anomalia identificada é grave o bastante para requerer o encerramento da sua execução, ou se uma execução com o sistema instável é admissível. O esforço despendido nesse item pode se refletir em: código de recuperação, cujo objetivo é restaurar o sistema a um estado válido; código de encerramento do sistema, cujo objetivo é finalizar a execução de forma a preservar ao máximo a integridade dos dados; e sistemas redundantes de hardware/software de maneira a não interromper a disponibilidade de serviços.

[3] Tratamento de falhas é a recuperação do sistema (após a ocorrência de uma falha) da forma mais elegante possível.

Esforço de remoção de faltas: É o esforço despendido para identificar e eliminar as faltas que levaram às falhas detectadas em ambiente de produção. O primeiro passo é identificar a versão do sistema onde se verificou a falha e, a partir da avaliação dos sintomas observados, tentar detectar a(s) real(is) causa(s) do problema. Uma vez detectada a causa, é necessário decidir se esta realmente deve ser removida pois, em alguns casos, a remoção de uma falta pode ser muito custosa em relação ao impacto causado ao sistema. Alternativas de coexistência com uma falta podem ser adotadas desde que a "filosofia" de minimização de suas conseqüências seja obedecida. Independente da decisão tomada (remoção da falha ou coexistência com a falha), se houver alteração de código é preciso fazer com que as correções cheguem o mais rápido possível aos usuários.

Embora grande parte do esforço ocorra com o sistema em ambiente de produção, é em tempo de desenvolvimento que devem ser desenvolvidas as ferramentas para que o tempo médio de remoção de falhas seja minimizado. Essas ferramentas muitas vezes se refletem em código cujo objetivo é obter o máximo de informações acerca do estado geral da execução do sistema no momento em que foi detectada a falha, de forma a facilitar a identificação da causa a partir do sintoma observado.

O correto balanceamento dos esforços despendidos em cada uma das quatro questões abordadas é o fator principal no desenvolvimento de sistemas orientados à recuperação. Não adianta concentrar esforços em um item em detrimento de outro, pois o retorno prático será insuficiente.

2.5 Agentes de Software

Um agente de software pode ser definido como um objeto complexo com atitude.[2] Um agente de software é governado pelo seu estado e seu comportamento. O estado do agente é descrito pelo seu conhecimento e expressado através de componentes mentais como crenças, objetivos e planos. O seu comportamento é composto e governado por uma série de características comportamentais, chamadas propriedades de agência.[18] Um agente pode possuir as seguintes propriedades:

- Autonomia
- Capacidade de reação

- Adaptação
- Interação
- Níveis de conhecimento
- Habilidade de comunicação
- Capacidade de inferência
- Comportamento colaborativo
- Continuidade temporal
- Personalidade
- Mobilidade

Essas propriedades e suas descrições podem ser encontradas em [19]. Dentre estas, existem três que são necessárias para que uma entidade seja considerada um agente de software:

Autonomia: objetivos direcionados, pró-atividade e iniciativa

Interação: habilidade de trabalhar em conjunto com outros agentes.

Adaptação: habilidade de aprender e evoluir com a experiência.

Agentes de software também podem ser classificados quanto ao seu grau de inteligência. Segundo Bradshaw[20], agentes com uma maior capacidade de cognição são classificados como agentes cognitivos, ao passo que os agentes menos cognitivos são classificados como agentes reativos. É interessante notar que agentes podem ter tanto características reativas como cognitivas - agentes puramente cognitivos ou reativos são apenas os dois extremos de um eixo imaginário onde podem ser posicionados todos os agentes de software já construídos ou ainda por construir. Idealmente, um agente de software que execute continuamente em um ambiente por um longo período de tempo deveria ser capaz de aprender com suas experiências. Adicionalmente, espera-se que um agente que habita um ambiente com outros agentes e processos seja capaz de comunicar e cooperar com eles, e talvez se mover de um lugar para outro ao fazer isso.

2.6 *Design by Contract*

Segundo Payne[21], a adoção precoce de algumas atividades no ciclo de desenvolvimento de um sistema é capaz de facilitar a criação e a realização de testes do sistema. Esta prática, conhecida por *design for testability* (DFT), é bastante difundida no mundo do desenvolvimento de hardware e se baseia na criação de testes embarcados e de medições de parâmetros pré-estabelecidos, de forma a facilitar a identificação e a correção de anomalias em tempo de desenvolvimento. O DFT é comumente usado no desenvolvimento de componentes de hardware complexos na tentativa de garantir a sua correta operação antes da etapa de produção em série, pois os custos de correção de uma anomalia após esta etapa são proibitivos. Um exemplo da aplicação do conceito do DFT é o uso do *Design by Contract*.

Segundo Meyer[1] o *Design by Contract* (DBC) é uma metodologia de desenvolvimento de software. A idéia central do DBC é que deve existir um contrato entre um artefato de software (classe, módulo, etc.) e seus clientes. O cliente deve garantir certas condições antes de invocar funcionalidades do artefato (como um método) e, depois da execução, o artefato deve garantir que certas condições ou propriedades serão verdadeiras.

O uso de pré e pós-condições na especificação de um software não é uma novidade. A grande diferença é que no DBC esses contratos são executáveis. Os contratos são definidos usando a própria linguagem de programação ou usando uma meta-linguagem (como JML[22] no caso de Java[23] ou a meta linguagem criada por Rosenblum[24] no caso de C++), e são convertidos em código pelo compilador (ou por um pré-compilador no caso das meta-linguagens). Dessa forma, qualquer violação de contrato que ocorra durante a execução do programa poderá ser detectada automaticamente [25].

O conceito de DBC foi criado com o objetivo de fornecer aos desenvolvedores de software um modo de produzir, a um baixo custo, software com um alto grau de confiabilidade. Baseado em uma simples metáfora, o DBC tem aplicações durante todo o processo de desenvolvimento de um software, da análise e design a implementação, documentação, depuração e até gerenciamento do projeto.[1]

Para implementar o DBC recomenda-se seguir algumas normas. A seguir são apresentados os 6 princípios básicos para uma boa implementação de DBC:[26]

– Separar *queries* de comandos. *Queries* retornam um resultado mas não mudam propriedades visíveis dentro de um objeto. Comandos podem

mudar um objeto mas não retornam valor.

- Separar as *queries* básicas das *queries* derivadas. *Queries* derivadas são especificadas a partir de *queries* básicas.
- Para cada *query* derivada, escrever uma pós-condição que especifique os resultados que serão retornados a partir de uma ou várias *queries* básicas. Dessa forma, se os valores das *queries* básicas são conhecidos, os valores das *queries* derivadas também são.
- Para cada comando, escrever a pós-condição que especifica o valor de cada *query* básica. Assim conhece-se o efeito completo de um comando.
- Para cada *query* e comando escolher um conjunto de pré-condições apropriado. As pré-condições determinam quando o cliente pode invocar *queries* e comandos.
- Escrever invariantes para definir as propriedades dos objetos que devem permanecer inalteradas. Concentrar-se nas propriedades que ajudam o programador a construir um modelo conceitual apropriado da abstração representada pela classe.

Segundo Mitchell[26], embora não haja estudos científicos que comprovem os benefícios da adoção do DBC em projetos de software, existem fortes indícios, coletados através de relatos de profissionais que fizeram uso desta tecnologia, que o DBC propicia melhorias como:

- Aumento da confiabilidade dos sistemas.
- Refinamento do processo de depuração.
- Os projetos passam a ter mais semântica, tornando-se mais claros e simples.
- Maior clareza e objetividade na documentação dos sistemas.
- Dentre outros.

2.7 *Bluetooth*

Bluetooth[27] é uma especificação aberta (*royalty-free*) de uma tecnologia para comunicação sem fio *ad hoc*, de curto alcance e baixo custo, através de conexões de rádio. Esta especificação se fortaleceu após 1998, quando cinco importantes empresas (Ericsson, Nokia, IBM, Intel e Toshiba) formaram o consórcio *Bluetooth SIG* (Special Interest Group) com o objetivo de expandir e promover a tecnologia *Bluetooth* e estabelecer um novo padrão industrial. A escolha do nome é uma homenagem ao unificador da Dinamarca, o rei Harald Blatand, mais conhecido como Harald *Bluetooth*

O padrão *Bluetooth* visa facilitar as transmissões de voz e dados em tempo real, assegurando proteção contra interferência e segurança dos dados transmitidos. Por meio da especificação *Bluetooth*, é possível conectar uma ampla variedade de dispositivos fixos (PC, impressora, mouse, teclado, scanner, etc.) e móveis (notebook, PDA, telefone celular, etc.) de uma forma bastante simples, sem a necessidade de utilizar cabos físicos de ligação. A idéia é permitir a interoperabilidade entre os dispositivos de forma automática, para que o usuário não precise se preocupar com isso.[27]

Alguns exemplos da aplicabilidade do *Bluetooth* são:

- Conexão sem-fio entre PC ou notebook e impressoras, scanners e a rede local. Conexão, também sem-fio, para o mouse e o teclado.
- O celular de uma pessoa pode saber automaticamente quando se encontra perto do notebook do seu dono, podendo assim enviar-lhe as mensagens de correio eletrônico recebidas, de forma transparente para o ser humano.
- Um dispositivo *Bluetooth* funcionando como um identificador pessoal de um usuário pode se comunicar com outros dispositivos *Bluetooth* em sua residência. Por exemplo, ao chegar em casa, a porta automaticamente se destrava para o usuário e as luzes são acesas.
- Um dispositivo *Bluetooth* contendo informações pessoais de um usuário pode funcionar com uma carteira de dinheiro eletrônica. Ao fazer compras, uma registradora desconta o valor da mercadoria adquirida.

Dispositivos *Bluetooth* operam na faixa ISM (Industrial, Scientific, Medical) centrada em 2,45 GHz. Esta faixa era formalmente reservada para alguns grupos de usuários profissionais, mas recentemente tem sido aberta mundialmente para uso comercial.[28] Nos Estados Unidos, a faixa ISM varia entre 2400 e 2483,5 MHz. No Brasil e na maior parte dos países europeus a

mesma banda também está disponível. No Japão a faixa varia entre 2400 e 2500 MHz.[29]

Devido à existência de diferentes regulamentações, existem iniciativas para uma padronização do espectro de freqüência da faixa ISM, objetivando assegurar uma compatibilidade mundial de comunicações.

Atualmente, o consórcio *Bluetooth* SIG já conta com a participação de cerca de 1400 empresas de todo o mundo. Este consórcio cresceu rapidamente com o suporte de companhias líderes em telecomunicações, eletrodomésticos e PCs interessadas no desenvolvimento de produtos baseados na nova especificação. Já fazem parte do consórcio empresas como 3Com, Compaq, Dell, HP, Lucent, Motorola, NTT DoCoMo, Philips, Samsung, Siemens, Texas, Microsoft, dentre outras.

A arquitetura do *Bluetooth* e suas características técnicas estão definidos nas especificações denominadas Core (Núcleo) e Profiles (Perfis), definidas pelo SIG. Enquanto a especificação do núcleo define como o sistema funciona (protocolos, camadas, especificações técnicas, etc.), o documento que define os perfis determina como os diversos elementos que compõem o sistema podem ser empregados para a realização das funções desejadas. Ao contrário de outros padrões, a especificação do *Bluetooth* compreende não apenas as camadas mais baixas da rede, mas também a camada da aplicação.

Apesar de muitos PCs e notebooks já incorporarem a tecnologia *Bluetooth* de fábrica, existe a possibilidade de incorporar tal tecnologia através de adaptadores. Os mais comuns são os de porta serial, de porta paralela e de USB. Seu uso é simples: é preciso conectar o adaptador ao PC através da porta definida e fazer uso do *driver* fornecido pelo fabricante do adaptador.

2.8 Biblioteca Log4Cxx

O log4cxx[30] é uma biblioteca *open source* que implementa um *log*[4] para C++ e se propõe a ser uma versão C++ da biblioteca Java log4j[31] - uma das bibliotecas de *log* mais utilizadas em programas Java. Tanto o log4j quanto o log4cxx são projetos da *Apache Software Foundation*[32].

No log4cxx (e no log4j) a configuração do *log* é feita a partir de um arquivo de configuração (um arquivo de propriedades ou um arquivo xml) lido no início da execução. Um trecho de código que configura o log4cxx pode ser visto no

[4] *Log* é o registro de informações sobre o que o programa fez ou está fazendo. É utilizado para acompanhar a execução do programa e auxilia na descoberta das causas de erros e mau funcionamento.

Código Fonte 2.1. Com o uso do arquivo de configuração é possível alterar toda a configuração do *log* sem que o programa precise ser recompilado. Na tentativa de ser o mais parecido possível com o log4j, o arquivo de configuração do log4cxx é idêntico a um arquivo de configuração do log4j. Isso permite que um arquivo usado em uma aplicação Java com log4j possa ser reutilizado em uma aplicação C++ com log4cxx e vice-versa (considerando que existem algumas funcionalidades/classes muito específicas ou complexas que ainda não estão disponíveis no log4cxx). Também é possível configurar o *log* através de código, mas esta não é considerada uma boa prática, pois além de deixar parte da configuração fixa (o programa teria que ser recompilado para alterá-la), ainda interfere no uso da configuração por arquivo, uma vez que pode sobrescrevê-la.

```
1  ...
2  using namespace log4cxx;
3  ...
4  int main(int argc, char **argv)
5  {
6      if(QFile::exists("log4cxx.properties"))
7      {
8          //configurando o log4cxx a partir de um arquivo de propriedades
9          PropertyConfigurator::configure("log4cxx.properties")
10     }
11     else if(QFile::exists("log4cxx.xml"))
12     {
13         //configurando o log4cxx a partir de um arquivo xml
14         DOMConfigurator::configure("log4cxx.xml")
15     }
16     else
17     {
18         //Utilizando a configuração básica do log4cxx
19         //(As mensagens são escritas no console e o Level do log é DEBUG)
20         BasicConfigurator::configure();
21     }
22     ...
23 }
```

Código Fonte 2.1: Configuração do log4cxx

Logger é a classe básica do log4cxx (e do log4j). Para registrar uma mensagem é preciso criar uma instância de um *Logger* e a partir dessa instância invocar seus métodos, que são: *debug*, *info*, *warn*, *error* e *fatal* (Código Fonte 2.2). Cada *Logger* é identificado por um nome e é configurado com um *Level* e um conjunto de *Appenders*.

O nome do *Logger* serve para determinar a hierarquia entre os *Loggers* criados. Quando um *Logger* é criado, herda as configurações do *Logger* pai na hierarquia. Por exemplo, o *Logger* chamado "br.com.util.Utilities" herda as configurações do *Logger* "br.com.util", o *Logger* "br.com.util" herda do "br.com", o "br.com" herda do "br"e o "br" herda do *Logger* raiz (que é pai de todos os *Loggers* e tem o nome de "root")(Figura 2.1). Este sistema permite que se configure apenas o *Logger* raiz, e essa configuração seja estendida a todos os outros *Loggers*. Também é possível configurar um grande grupo de

```
1  ...
2  using namespace log4cxx;
3  ...
4  const Logger* TestLog::LOG = Logger::getLogger("br.com.test.TestLog");
5  ...
6  {
7      ...
8      //DEBUG
9      if(LOG->isDebugEnabled())
10     {
11         LOG->debug("mensagem de level debug", __FILE__, __LINE__);
12     }
13     //ou
14     LOG4CXX_DEBUG(LOG, "mensagem de level debug");
15
16     //INFO
17     if(LOG->isInfoEnabled())
18     {
19         LOG->info("mensagem de level info", __FILE__, __LINE__);
20     }
21     //ou
22     LOG4CXX_INFO(LOG, "mensagem de level info");
23
24     //WARN
25     if(LOG->isWarnEnabled())
26     {
27         LOG->warn("mensagem de level warn", __FILE__, __LINE__);
28     }
29     //ou
30     LOG4CXX_WARN(LOG, "mensagem de level warn");
31
32     //ERROR
33     if(LOG->isErrorEnabled())
34     {
35         LOG->error("mensagem de level error", __FILE__, __LINE__);
36     }
37     //ou
38     LOG4CXX_ERROR(LOG, "mensagem de level error");
39
40     //FATAL
41     if(LOG->isFatalEnabled())
42     {
43         LOG->fatal("mensagem de level fatal", __FILE__, __LINE__);
44     }
45     //ou
46     LOG4CXX_FATAL(LOG, "mensagem de level fatal");
47
48     /*
49     Os ifs não são realmente necessários, pois o próprio método que registra a
           mensagem testa se o determinado level está habilitado. Os ifs
           simplesmente evitam a criação desnecessária da mensagem (que pode ser
           mais complexa que uma simples string, utilizando por exemplo
           concatenação de strings e acesso a métodos para complementar os dados
           da mensagem) e dos dois outros parâmetros, e a invocação desnecessária
           de um método com três parâmetros.
50     */
51     /*
52     __FILE__ e __LINE__ são macros pré definidas de C++ que em tempo de
           compilação são substituídas respectivamente, pelo nome do arquivo (um
           const char*) e pela linha corrente (um int).
53     */
54     ...
55 }
```

Código Fonte 2.2: Registrando mensagens no log4cxx

Loggers ao mesmo tempo apenas configurando um *Logger* pai que seja comum entre eles (por exemplo, pode-se configurar o *Logger* "br.com.util" para alterar a configuração de todos os *Loggers* cujos nomes começam com "br.com.util.")

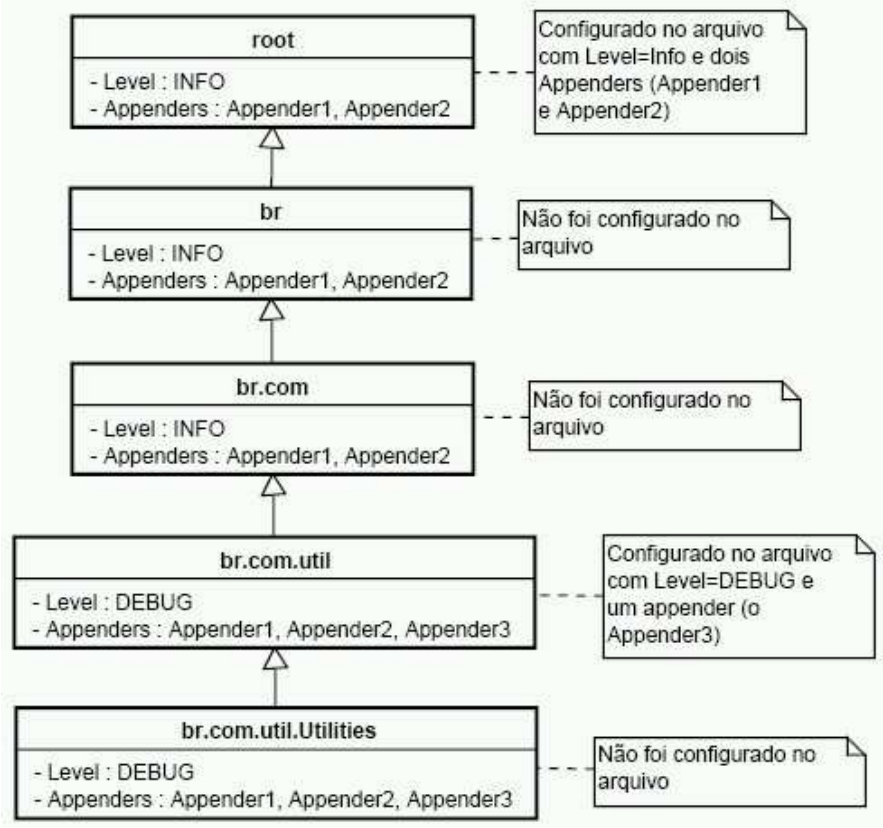

Figura 2.1: Herança dos *Loggers* - Como alguns *Loggers* estariam configurados se apenas os *Loggers* "root" e "br.com.utlites" fossem configurados no arquivo.

O *Level* define quais mensagens estarão sendo registradas. Os *Levels* são em ordem de prioridade (do menor para o maior) *DEBUG*, *INFO*, *WARN*, *ERROR* e *FATAL*. Um *Logger* configurado para *WARN* por exemplo não irá registrar nenhuma mensagem de *INFO* ou *DEBUG*. Um *Logger* também pode ser desabilitado.

Os *Appenders* definem a forma como as mensagens serão registradas. Existem *Appenders* para os mais variados formatos: console, arquivo, e-mail, banco de dados, etc. Um mesmo *Logger* pode ter vários *Appenders* distintos. Assim como os *Loggers*, os *Appenders* também podem ser configurados com um *Level*, ou seja, uma determinada mensagem pode ser registrada apenas por alguns *Appenders* de um *Logger*. Por exemplo, um *Logger* pode ter um *Appender* que escreve em um arquivo e um *Appender* que envia e-mails, sendo que, o *Appender* de e-mail pode estar configurado para só registrar mensagens de *Level ERROR* (ou maior), enquanto o *Appender* de arquivo pode estar configurado para

registrar mensagens de *Level INFO* (ou maior). Os *Appenders* possuem ainda um *Layout* que define como a mensagem será formatada. Um mesmo *Appender* pode ser utilizado por vários *Loggers*, permitindo configurar um determinado *Appender* e reutilizá-lo na configuração de vários *Loggers*.

As boas práticas recomendam que cada classe tenha o seu próprio *Logger* e que ele seja declarado como uma constante privada e estática, pois assim cada classe possuirá apenas um *Logger* e apenas a própria classe poderá acessá-lo (normalmente essa constante recebe o nome de *LOG*). Recomenda-se também que o nome do *Logger* seja no formato "<prefixo>.<Nome da Classe>", onde este prefixo seja algo que identifique o módulo, pacote ou o grupo de classes ao qual a classe pertence como por exemplo: "io", "io.net", "db", "model", etc (em Java costuma-se usar como o nome do *Logger* o nome completo da classe, ou seja "<nome do pacote>.<nome da classe>"). O uso deste padrão para os nomes simplifica a alteração da configuração do *log* através do uso das hierarquias (já comentadas anteriormente). Apesar desta ser a maneira mais comum de se usar o log4cxx (e o log4j), existem aplicações que só podem dispor de uma quantidade limitada de memória, onde utilizar um *Logger* para cada classe do sistema pode ser inviável. Nestes casos ao invés de um *Logger* por classe pode-se usar um *Logger* por grupo de classes. Uma política nesse sentido seria, por exemplo, criar um Logger para cada pacote: um *Logger* para todas as classes de io; um *Logger* para todas as classes de banco de dados; etc.

Uma questão a ser observada é que o uso do *log* implica em um custo durante a execução. No entanto, enquanto registrar uma mensagem é uma ação que consome recursos (tempo, memória, etc.), testar se uma mensagem deve ou não registrada tem um custo insignificante (na maior parte das vezes este teste consiste na invocação de uma função que faz apenas um teste booleano - se o *Level* da mensagem é maior ou igual ao *Level* do *Logger*). Este fato pode viabilizar o uso do *log* mesmo em aplicações que possuem requisitos de performance. A aplicação pode ser desenvolvida com todas as mensagens habilitadas e a versão final pode ser distribuída com o *log* configurado para que apenas as mensagens realmente importantes (como as mensagens de erro) sejam registradas.

2.9 Biblioteca SQLite

SQLite[33] é uma biblioteca *open source* desenvolvida em C que implementa um banco de dados auto-contido, ou seja, implementa todas as funcionalidades de um banco de dados sem depender de nenhum outro software

ou biblioteca. Ele não necessita de instalação ou configuração, o que pode ser uma grande vantagem em determinadas aplicações. Tais características permitem a inclusão do SQLite em aplicações que façam uso do mesmo sem que haja necessidade de qualquer tipo de instalação ou configuração extra.

O projeto SQLite existe desde maio de 2000, sendo bastante utilizado pela comunidade *open source*. Em 2005 o projeto SQLite e seu principal autor, D. Richard Hipp, ganharam o prêmio "*Open Source Awards*"[34, 35] na categoria *Integrator*. O Open Source Awards é um prêmio oferecido pelo Google e pelo O'Reilly[36] aos projetos *open source* de maior destaque no ano, onde são premiados cinco projetos em diferentes categorias (*Communicator*, *Evangelist*, *Diplomat*, *Integrator* e *Hacker*).

O SQLite armazena todos os dados em um único arquivo, o qual acessa diretamente. Este arquivo pode ser manipulado como um arquivo comum - o que facilita a realização de cópias, backups e distribuição da base - e pode ser compartilhado entre máquinas com diferentes *byte orders* (*little endian*/*big endian*).

Apesar de ser pequeno e leve (250KB), o SQLite implementa praticamente todo o SQL92 (tendo menos de uma dezena de comandos não suportados), suporta bases com até 2 terabytes, suporta *strings* e *BLOBs* de tamanho ilimitado (limitado apenas pela memória disponível) e é em geral mais rápido que os banco de dados cliente/servidor mais populares.[33] Outra característica importante do SQLite é que ele implementa não só o banco de dados, mas também a camada de persistência (o código que faz acesso ao banco e manipula o resultado das consultas).

A execução de comandos SQL no SQLite é feita através de chamadas à função *sqlite3_exec*. O Código Fonte 2.3 apresenta a assinatura desta função.

```
1 int sqlite3_exec(    sqlite3*          db,
2                      const char*       sqlCommand,
3                      sqlite_callback   callbakcFunction,
4                      void*             callbackAdditionalParameter,
5                      char**            errorMessagePtr);
```

Código Fonte 2.3: Função sqlite3_exec

A função *sqlite3_exec* retorna um código (int) indicando que o comando foi executado com sucesso ou um código de erro[5]. Ela recebe os seguintes parâmetros:

db: Banco de dados (uma estrutura própria do SQLite, o *sqlite3*).

[5]Como é comum em funções de C/C++ o código de retorno para um sucesso é 0, qualquer outro valor significa um erro. O SQLite tem uma lista pré-definida com códigos relacionados aos erros mais comuns.

sqlCommand: Comando SQL (um *const char* *).

callbakcFunction: Função de callback (será detalhada a seguir; pode ser NULL).

callbackAdditionalParameter: Parâmetro extra para a função de callback (um *void**; pode ser NULL).

errorMessagePtr: Ponteiro para receber a mensagem de erro (um *char***; pode ser NULL).

A função de callback é a forma usada pelo SQLite para manipular o resultado das consultas. Esta função é executada uma vez para cada linha retornada como resultado de uma consulta SQL. A função recebe em cada execução os parâmetros correspondentes a uma determinada linha. É dentro da função de callback que o resultado da consulta pode ser manipulado, o que permite, por exemplo, o teste dos valores retornados ou a criação de objetos e/ou estruturas de dados. Qualquer função pode ser usada como função de callback, desde que possua a assinatura apresentado no Código Fonte 2.4.

```
1 int callBackFunction(    void *  additionalParameter,
2                          int     columnCount,
3                          char ** columnNames,
4                          char ** columnValues)
```

Código Fonte 2.4: Exemplo de função de callback

Assim como a função *sqlite3_exec* a função de callback deve retornar um código (int) indicando que o comando foi executado com sucesso ou um código de erro (os códigos são os mesmo que podem ser retornados pela sqlite3_exec). Ela recebe os seguintes parâmetros:

additionalParameter: Parâmetro extra passado na chamada da função *sqlite3_exec* (o 4° parâmetro da *sqlite3_exec*)(um *void**).

columnCount: Número de colunas da linha corrente (um *int*).

columnNames: Array com o nome das colunas da linha corrente (um *char***, é um array de tamanho igual ao parâmetro *columnCount*).

columnValues: Array com o valor de cada coluna da linha corrente (um *char***, é um array de tamanho igual ao parâmetro *columnCount*).

2.10 Biblioteca Qt

Qt[37] é uma biblioteca escrita em C++. Ele foi inicialmente criado para ser usado no desenvolvimento de aplicações GUI (*Graphical User Interface*), no entanto, este já deixou de ser seu foco principal, e hoje o Qt é uma biblioteca de uso geral. Ele possui um conjunto grande e bem estruturado de classes (mais de quatrocentas) que dão suporte a um grande número de funcionalidades, e possui desde classes básicas (*strings*, coleções, *maps*, arquivos, etc.) até classes mais específicas (GUI, XML, internet, internacionalização, openGL, etc.).

Através do uso do Qt é possível escrever programas que podem ser compilados para diferentes plataformas a partir de um único código fonte. Portanto, o Qt pode ser considerado como uma alternativa ao Java[23] para a geração de sistemas multi-plataforma, com a diferença que em Java a compatibilidade está em nível de código objeto com a existência de um interpretador que é único por plataforma, ao passo que no Qt a compatibilidade está em nível de código fonte, sendo necessário gerar uma compilação diferente para cada plataforma.

Uma das principais características do Qt é o mecanismo de comunicação entre objetos realizado através dos *slots* e *signals*, que será aprofundado na Seção 2.10.1.

Uma desvantagem do Qt é que para gerar o código responsável pelos *slots* e *signals*, ele precisa pré-processar os arquivos que fazem uso dessas funcionalidades antes do código ser compilado. Esse pré-processamento gera mais dois arquivos (um .h e um .cpp - Figura 2.2) com o código extra. Devido a este fato há um aumento significativo no tempo de compilação de um projeto que faz uso do Qt (o processo de gerar os arquivos extras é bem rápido, mas como este processo cria um novo arquivo .cpp que também precisa ser compilado, o tempo total de compilação de cada classe aumenta consideravelmente). Além disso, o fato do pré-processamento gerar/alterar um arquivo .h faz com que possa ocorrer uma recompilação em massa[6], mesmo quando apenas o arquivo .cpp é alterado.

[6]Recompilação em massa, é a necessidade de recompilar arquivos que não foram alterados apenas porque eles dependem de arquivos que foram alterados. Normalmente em um projeto C/C++ alterar um .cpp (ou .c) não causa recompilação em massa. No caso do Qt, o pré-processamento do .cpp pode criar/alterar um .h, e isso sim pode provocar uma recompilação em massa. Sendo assim, a alteração de um arquivo .cpp que faz uso do Qt pode causar, mesmo indiretamente, uma recompilação em massa.

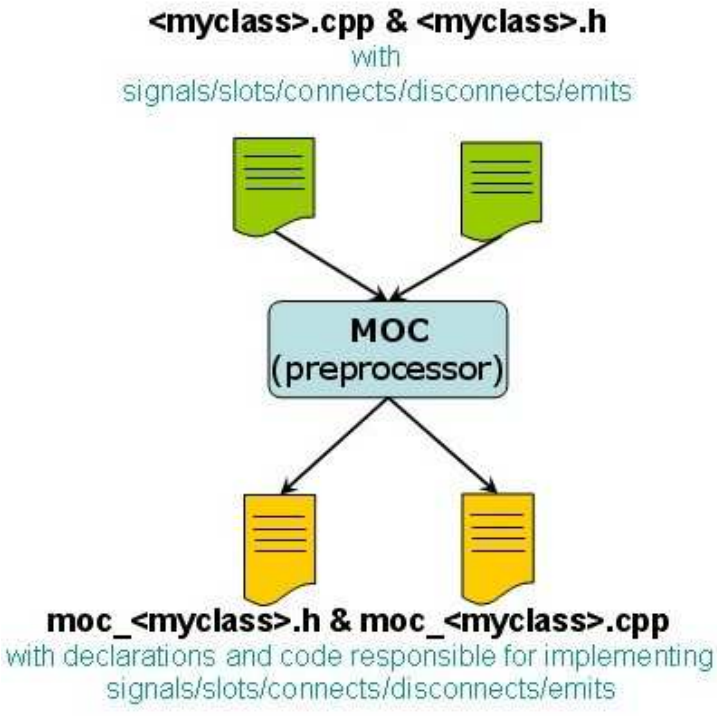

Figura 2.2: Pré-processamento do Qt[17]

2.10.1 *Signals* e *Slots*

O Qt possui um mecanismo próprio para implementar o tratamento de eventos e a comunicação entre objetos: os chamados *signals* e *slots*. Um *signal* é um sinal que um objeto emite quando um determinado evento ocorre. Um *slot* é uma função que um objeto executa quando um determinado *signal* é emitido.

O Qt permite a conexão de *slots* a *signals* (Figura 2.3). Assim, quando um objeto emite um *signal*, todos os *slots* conectados àquele *signal* são executados. Um dos grandes diferenciais deste mecanismo é que os *signals* e *slots* possuem um acoplamento bem fraco: o objeto que emite o *signal* não têm conhecimento dos *slots* que receberão o *signal* ou mesmo do objeto detentor do *slot* (não existe nenhuma lista de *listeners* ou *observers*), assim como quando um *slot* é executado, o objeto detentor do *slot* não conhece o objeto que emitiu o *signal* nem qual foi o *signal* emitido (no caso de um mesmo *slot* estar conectado a mais de um *signal*).

Para uma classe possuir *slots* e/ou *signals* ela deve estender da classe QObject - que é a classe básica do Qt (praticamente todas as classes do Qt são QObjects) - e deve possuir a macro Q_OBJECT em sua declaração. Nas declarações de cada classe são declarados os *signals* que aquela classe pode emitir, e as funções que podem ser usadas como *slots* (Código Fonte 2.5).

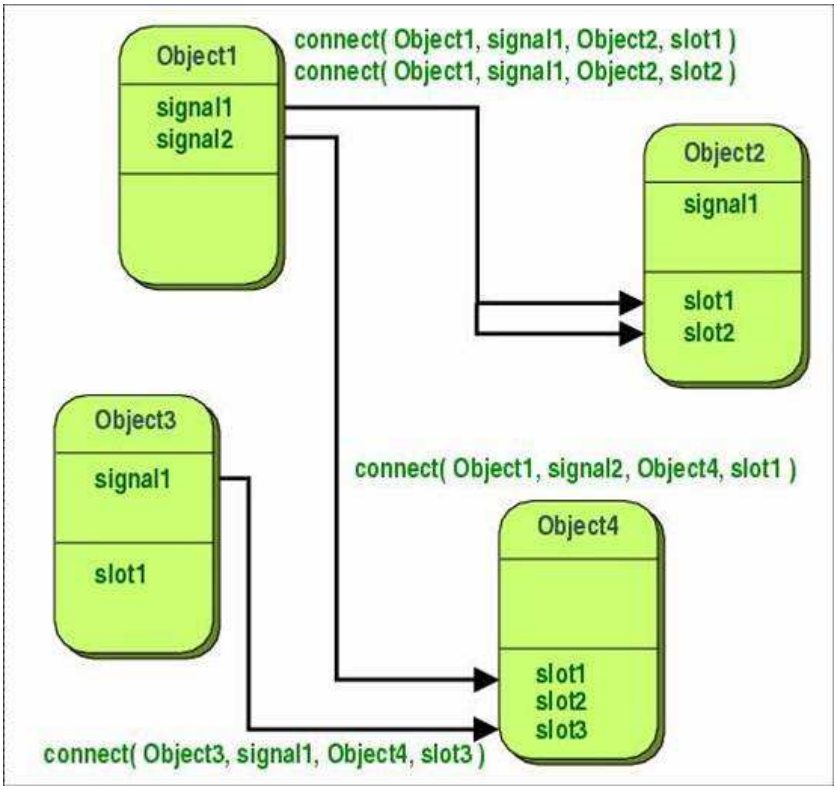

Figura 2.3: Conectando *slots* a *signals*[17]

```
1  #include <qobject.h>
2
3  class Counter : public QObject
4  {
5     Q_OBJECT
6     public:
7        Counter()
8        {
9           this->value = 0;
10       }
11       virtual ~Counter()
12       {
13       }
14       int getValue() const
15       {
16           return this->value;
17       }
18    public slots:
19       void setValue(int value);
20    signals:
21       void valueChanged(int newValue);
22     private:
23       int value;
24 };
```

Código Fonte 2.5: Exemplo QObject - *Signals* e *Slots*

Através do Código Fonte 2.6 é feita a conexão entre um *signal* e um *slot*. A desconexão é feita de forma similar (Código Fonte 2.7).

```
1 ...
2 Counter a;
3 Counter b;
4 ...
5 QObject::connect(&a, SIGNAL(valueChanged(int)), &b, SLOT(setValue(int)));
6 ...
```

Código Fonte 2.6: Exemplo Qt::connect

```
1 ...
2 Counter a;
3 Counter b;
4 ...
5 QObject::disconnect(&a, SIGNAL(valueChanged(int)), &b, SLOT(setValue(int)));
6 ...
```

Código Fonte 2.7: Exemplo Qt::disconnect

Depois que um *signal* de um objeto é conectado ao *slot* de outro objeto, sempre que aquele objeto emitir o determinado *signal* (Código Fonte 2.8) aquele *slot* especifico será executado.

```
1 void Counter::anyMethod()
2 {
3   ...
4   emit valueChanged(value);
5   ...
6 }
```

Código Fonte 2.8: Exemplo emitindo *Signal*

Existe uma restrição quanto aos *slots* e *signals*: só é possível conectar a um *signal*, os *slots* que tenham uma assinatura compatível. Por exemplo: o *signal* bar(int, QString, char) pode ser conectado a qualquer *slot* que não receba nenhum parâmetro, ou que receba como parâmetros um int e/ou um *QString* e/ou um char (necessariamente nessa ordem).

Vale ressaltar que um *slot* é um método normal, ou seja, ele pode ser invocado normalmente e pode ter os mesmo modificadores de um método normal (*public*, *private*, *protected*, *static*, *virtual*, etc.). Tais modificadores podem restringir os *signals* que podem ser conectados àquele *slot* (por exemplo: um *slot* declarado como *private* só pode ser conectado a *signals* da própria classe; e um *slot* declarado como *protected* só pode ser conectado a *signals* da própria classe ou de suas sub-classes).

2.11 *Mock Objects*

Mock Objects é um padrão de teste proposto por Mackinnon[3] onde um objeto é substituído por um objeto falso (o *Mock Object*) que o simulará durante os testes. Em outras palavras, um *Mock Object* é uma implementação falsa de um objeto, que possui a mesma interface do objeto real, e é usado para simular o objeto real exclusivamente para a realização de testes. *Mock Objects* são usados quando:

- O objeto real possui um comportamento não determinístico (produz resultados imprevisíveis).
- O objeto real é difícil de configurar ou instanciar.
- O objeto real tem um comportamento específico difícil de ser repetido (por exemplo: um erro de conexão).
- O objeto real consome muitos recursos (memória, tempo, rede, etc.).
- O objeto real não pode ser utilizado (por exemplo: o objeto real ainda está sendo desenvolvido; o objeto real só é acessível quando o programa é executado em um determinado ambiente que não pode ser acessado livremente; etc.).
- É preciso adicionar algum comportamento extra ao objeto real, sem alterá-lo (por exemplo: registrar no *log* cada chamada a métodos do objeto real). Neste caso ao invés de substituir o objeto real, o *Mock Object* se interpõe entre o sistema e o objeto real. O sistema passa a acessar o *Mock Object* que repassa todas as requisições do sistema para o objeto real. O *Mock Object* pode então realizar operações extras antes e depois de invocar os métodos do objeto real.
- O teste precisa de informações sobre a freqüência de uso do objeto real (por exemplo: quantas vezes um determinado método está sendo invocado).
- etc.

O seu uso mais comum é na obtenção de informações sobre uso de um objeto, onde o uso de *Mock Objects* permite definir dentro dos testes expectativas como:

- Quais métodos foram executados durante os testes.
- Quantas vezes cada método foi executado.
- Em que ordem os métodos foram executados

– Quais valores foram retornados ou passados como argumento em um método.

– etc.

Neste trabalho, os *Mock Objects* foram usados para simular partes do sistema que, ou estavam sendo desenvolvidas em paralelo por pessoas diferentes, ou iriam ser desenvolvidas no futuro.

Existem algumas APIs Java[23] que implementam o mecanismo de *Mock Objects*, como *Easy-Mock*[38], *JMock*[39] e *Mock Maker*[40]. No Código Fonte 2.9 é apresentado um exemplo de um caso de teste utilizando *JMock*. Este teste verifica se, durante a execução, o método *testMethod* do *Mock Object* foi executado uma (e apenas uma) vez com o parâmetro *"foo"* e retornou *"bar"*.

```
1  ...
2  import org.jmock.Mock;
3  import org.jmock.MockObjectTestCase;
4  ...
5
6  public class ExampleTest extends MockObjectTestCase
7  {
8      public void testMethod()
9      {
10         ExampleTest mockExample = new ExampleTest();
11
12         // Step 1: create mock
13         Mock mockObj = mock(AnyClass.class);
14         // Step 2: set expectations (as part of the test).
15         mockExample.expects(once()).method("testMethod")
16                                    .with( eq("foo") )
17                                    .will(returnValue("bar"));
18         // Step 3: call method to be tested and pass mock as argument
19         Client client = new Client();
20         client.method(mockObj);
21         // Step 4: verify expectations
22         mockExample.verify();
23     }
24 }
```

Código Fonte 2.9: Exemplo de caso teste com Mock Objects utilizando JMock

2.11.1 Mock Objects X Stubs

À primeira vista, um *Mock Object* pode parecer apenas um simples *stub*. Uma análise mais detalhada demonstra, no entanto, que um *Mock Object* possui características que o diferenciam de um simples *stub*.[41]

Um *stub* normalmente é usado para retornar dados específicos quando um método é invocado com determinados parâmetros ou para executar pequenos testes (como alterar um valor booleano se um determinado método for invocado).

Um *Mock Object* é usado para obter informações sobre a execução de um determinado objeto ou para simular um objeto durante os testes, quando existe algum empecilho que dificulte ou impeça o uso do objeto real nos testes.

Apesar dessa diferença, ferramentas usadas para criar *Mock Objects* normalmente se mostram muito eficientes para escrever *stubs*.

2.12 Componentes de Software

O desenvolvimento baseado em componentes é atualmente um dos tópicos mais relevantes da Engenharia de Software. Ele surge como uma ponte, ou seja, como uma técnica que traz os recursos necessários para, a partir do entendimento do domínio de uma aplicação, poder dividi-la em componentes. A técnica postula que os componentes podem ser especificados por meio de funcionalidades e interfaces bem definidas, de forma que podem ser utilizados por terceiros que, não necessariamente, conheçam detalhes de sua implementação.[42]

Um componente de software pode ser definido como uma unidade de software independente, que encapsula, dentro de si, seu projeto e implementação, e oferece serviços para o meio externo, por meio de interfaces bem definidas. Os componentes se conectam por meio da interface requerida de um com a interface fornecida de outro.[4, 43]

A reutilização é uma das principais características de um componente. Os maiores benefícios da reutilização de software se referem ao aumento da qualidade e redução do esforço de desenvolvimento. O desenvolvimento para e com reutilização é uma tarefa complexa que tem conseqüências tanto técnicas quanto gerenciais e organizacionais. Segundo Souza[42] alguns dos benefícios relacionados com componentes e reutilização são destacados a seguir:

Redução de custo e tempo de desenvolvimento: Devido ao fato de evitar retrabalho ao desenvolvimento de componentes.

Gerenciamento de complexidade: Lidando com um número reduzido de componentes de cada vez, é possível gerenciar a complexidade do sistema, reduzindo, portanto, os riscos de desenvolvimento.

Desenvolvimento paralelo: A decomposição do sistema em componentes permite a definição de componentes independentes, que podem ser subcontratados, pelo menos em parte. Assim, o desenvolvimento pode se concentrar nas interfaces e conectores entre os componentes.

Aumento de qualidade: A reutilização leva ao aumento da qualidade dos componentes, uma vez que estes são previamente utilizados e testados. Todavia, isto também deve ser analisado com cuidado para garantir que os componentes foram utilizados e testados em ambientes compatíveis.

Facilidade de manutenção e atualização: O fato do sistema ser construído a partir de componentes facilita a localização de modificações e atualizações.

Por outro lado, o desenvolvimento baseado em componentes apresenta problemas que incluem:[42]

Seleção do componente certo: Encontrar o componente certo disponível é uma tarefa difícil. Muita pesquisa ainda precisa ser feita na organização e recuperação de bibliotecas de componentes.

Confiabilidade dos componentes: É necessário assegurar que os componentes foram utilizados e testados anteriormente em um ambiente compatível. Uma documentação, que inclua conjuntos de testes para cada componente, deve ser fornecida.

Custo e tempo de desenvolvimento: O desenvolvimento de software usualmente sofre pressões de tempo e custo. Da mesma forma que a reutilização pode ajudar (desenvolvimento com reutilização) também pode requerer esforço adicional. Esse esforço deve-se à possível reutilização futura dos componentes (desenvolvimento para reutilização) e à necessidade de serem flexíveis, estáveis e corretos. Tal esforço é consideravelmente superior ao necessário para qualquer outro software desenvolvido para uma aplicação específica.

Cultura dos engenheiros de software: Os engenheiros de software têm boa formação, são altamente criativos e gostam, portanto, de inventar soluções próprias. Conseqüentemente, são relutantes em confiar em código desenvolvido por terceiros e querem ter o controle do desenvolvimento do software.