5
Implementação

5.1
O Software Embarcado

O software embarcado foi desenvolvido em C. Ele é responsável por todas as tarefas de gerenciamento e controle dos processos internos da ferramenta de inspeção. Dentre elas, as mais importantes são: ler os dados coletados por sensores, empacotar tais informações e enviá-las ao software supervisor e verificar a correta execução do sistema através do monitoramento dos sensores (como temperatura e tensão na bateria).

Os sensores da ferramenta podem ser de várias naturezas, de acordo com a grandeza medida (campo elétrico, campo magnético, ultra-som, etc.), e são divididos em grupos. Cada grupo possui um conjunto de sensores dispostos em fila. Quando a ferramenta se movimenta sobre uma área do duto que contém uma anomalia (corrosão, amassado, furo) ou uma solda (as soldas são usadas como referência para as anomalias) ocorre uma alteração na grandeza medida, que é imediatamente detectada pelos sensores. A partir dessas alterações é possível determinar a presença de soldas e anomalias e a gravidade das anomalias.

O software embarcado foi definido como um sistema multi-agentes devido às suas características autônomas. Foram definidos quatro tipos de agentes com funcionalidades bem específicas. São eles o **Agente Comunicador**, o **Agente de Sensor**, o **Agente Odométrico** e o **Agente Guardião**. Cada um dos agentes, além da interação entre os mesmos, será melhor explicado a seguir.

O **Agente Comunicador** é responsável pela comunicação entre o software embarcado e o software supervisor. Ele é responsável por montar, a partir dos dados enviados pelos outros agentes, os pacotes de dados que serão enviados, e também por desmontar e interpretar os pacotes de dados recebidos. O **Agente Comunicador** não processa o conteúdo da mensagem que está enviando, sua tarefa resume-se a fazer os dados chegarem corretamente ao software supervisor, tratando questões como qualidade de serviço, consistência e integridade dos pacotes. Ele também é capaz de interpretar os pedidos oriundos do software

supervisor.

Cada **Agente de Sensor** monitora um grupo de sensores. Há agentes para sensores magnéticos, agentes para sensores geométricos, e assim por diante. Cada agente deve averiguar se os sensores estão funcionando corretamente, ler os dados de cada sensor e enviar tais dados ao **Agente Comunicador**.

O **Agente Odométrico** monitora o sensor odométrico da ferramenta, que é o responsável por fornecer informação de espaço percorrido. Um odômetro é um componente que se assemelha a um cilindro, e que tem um tambor interno que gira, gerando uma quantidade de pulsos constante a cada revolução. Sabendo-se qual o perímetro da superfície do odômetro, é possível calcular o deslocamento realizado a partir do número de pulsos contados. Com isso, é possível calcular com precisão as distâncias percorridas pela ferramenta. Ele também é responsável por ordenar que os **Agentes de Sensor** adquiram dados, de acordo com a precisão configurada.

O **Agente Guardião** (ou *Watch Dog*) é responsável por monitorar constantemente a execução do sistema, observando os sensores relacionados ao bom funcionamento do hardware e do software - de tempos em tempos ele consulta os demais agentes para a verificação de execução e, se for preciso, pode reiniciar um agente específico ou mesmo decidir pela reinicialização do sistema (um *reboot*).

O processo se inicia quando a ferramenta começa a se movimentar. Neste momento, os sensores já estão detectando as grandezas (campo magnético, campo elétrico, ultra-som, etc.). A cada intervalo percorrido (cujo valor é dependente da natureza dos sensores envolvidos mas, normalmente, varia entre 1 e 5 milímetros), o **Agente Odométrico** envia uma requisição aos **Agentes de Sensores**, para que eles leiam a grandeza às quais se referem. Cada **Agente de Sensor** deve ler os dados de cada sensor e enviar tais dados ao **Agente Comunicador**. O diagrama de colaboração dos agentes pode ser visto na Figura 5.1.

É importante observar que o paradigma de agentes de software foi usado na modelagem e no projeto do software embarcado, mas não na implementação propriamente dita, devido às limitações existentes no desenvolvimento de software desta natureza onde normalmente nem mesmo o uso de OO é possível. No entanto a utilização de agentes na modelagem do sistema permitiu uma melhor compreensão em relação ao comportamento do sistema, em como ele interage com o software supervisor, e como cada parte do sistema interage com as outras partes.

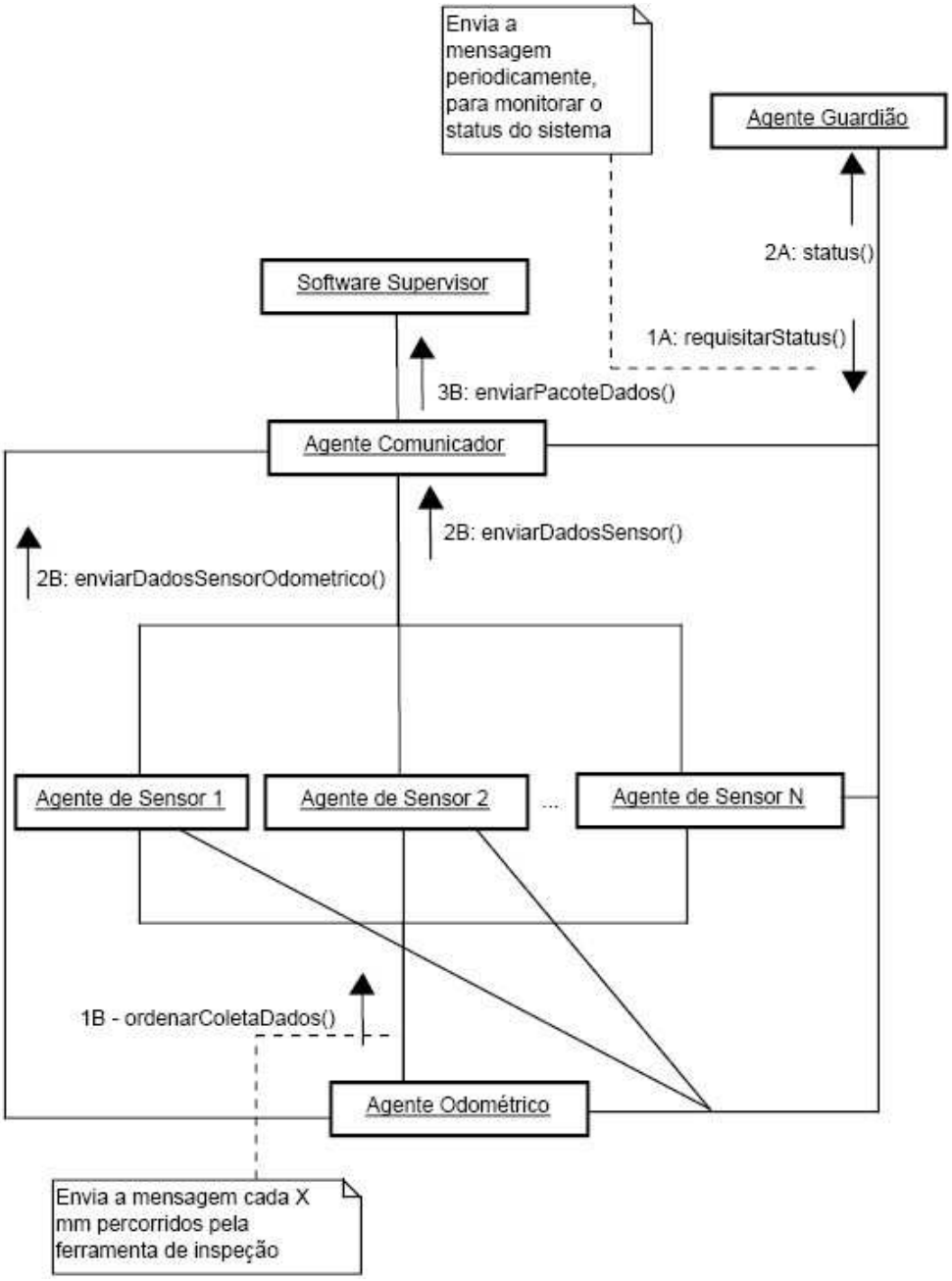

Figura 5.1: Diagrama de colaboração dos agentes - Software Embarcado

5.2 O Software Supervisor

O software supervisor foi desenvolvido em C++. Ele é responsável por permitir a visualização e a análise dos dados que são recebidos do software embarcado.

O software supervisor foi definido como uma sistema multi-agentes onde existem três agentes (**Agente Comunicador**, **Agente Interpretador** e **Agente Documentador**) e três módulos acessórios (**Módulo de Estatísticas**, **Módulo de Visualização** e **Módulo de Banco de Dados**).

O **Agente Comunicador** é responsável pela comunicação entre o software supervisor e o software embarcado. Ele é responsável por enviar e receber os pacotes de dados. O **Agente Comunicador** não conhece o conteúdo da mensagem que está enviando, sua tarefa resume-se a enviar e receber os dados, tratando questões como qualidade de serviço, consistência e integridade dos pacotes. A cada tentativa de comunicação com o software embarcado, bem sucedida ou não, o **Agente Comunicador** atualiza os dados do **Módulo de Estatísticas**.

O **Agente Interpretador** é responsável por empacotar e desempacotar os pacotes que o **Agente Comunicador** envia ou recebe. Quando alguma mensagem precisa ser enviada, um comando é enviado ao **Agente Interpretador** que gera um pacote de dados e o entrega ao **Agente Comunicador**, que o enviará. Da mesma forma quando o **Agente Comunicador** recebe um pacote e o entrega ao **Agente Interpretador** que decodifica o pacote e o envia ao agente **Agente Documentador** (ou ao **Módulo de Visualização**).

O **Agente Documentador** faz a análise e o armazenamento dos dados capturados pelos sensores da ferramenta de inspeção. Ele é responsável por converter os dados lidos dos sensores, os quais são medidos numa escala em volts (no caso dos sensores magnéticos) ou pulsos (no casos dos odômetros), em dados que podem ser analisados pelos usuários do sistema. Para isso, faz uso de funções conversoras, as quais aplicam constantes de calibração a fim de obter, a partir da voltagem lida ou do número de pulsos, a grandeza física correspondente na unidade de engenharia correta (Gauss, Metros, etc.). Sempre que novos dados são recebidos o **Agente Documentador** atualiza os dados do **Módulo de Visualização**.

O **Módulo de Estatísticas** armazena informações sobre a comunicação com o software embarcado, permitindo obter informações como: tempo de conexão, número de falhas na comunicação, número de pacotes inválidos, percentual de pacotes válidos, etc. Estes dados são utilizados na análise da

qualidade da comunicação, uma vez que a comunicação por *bluetooth* pode sofrer interferências.

O **Módulo de Visualização** é responsável pela interface gráfica do sistema, o que inclui apresentar os dados, sejam eles dados recebidos do software embarcado ou dados oriundos da base de dados, e tratar as requisições da interface.

O **Módulo de Banco de Dados** é responsável por fazer o acesso ao banco de dados do sistema, armazenando, removendo ou recuperando dados. Este módulo é totalmente baseado na biblioteca SQLite[33].

O diagrama de colaboração dos agentes juntamente com os módulos pode ser visto na Figura 5.2.

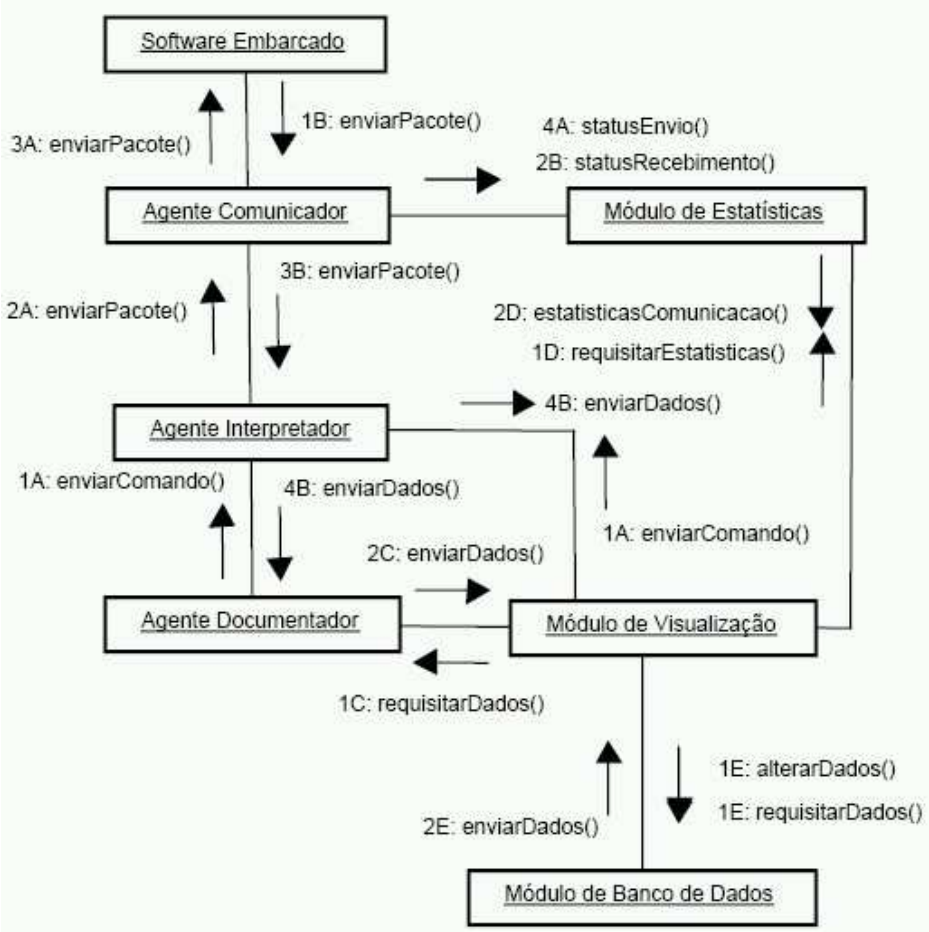

Figura 5.2: Diagrama de colaboração dos agentes - Software Supervisor

A comunicação entre o software embarcado e o software supervisor é assíncrona: existe o método *send* (no **Agente Comunicador** do software supervisor), que recebe um *array* de *bytes* e os coloca em uma fila de envio, retornando imediatamente. De forma semelhante, quando um novo dado chega, o **Agente Comunicador** toma a iniciativa de avisar, através da emissão de um

signal. Para isso, existe um *Thread* separado para o **Agente Comunicador**, que fica constantemente verificando a fila de envio e a fila de recepção de dados. Dessa forma, os demais *Threads* do sistema não ficam travados à espera do envio ou do recebimento de dados.

Existem três *signals* que podem ser levantados pelo **Agente Comunicador**:

New Sample: Emitido quando o **Agente Comunicador** recebe uma amostra válida do software embarcado.

Reply to a previous request: Emitido quando o **Agente Comunicador** recebe uma resposta a um comando que fora enviado previamente.

Request sent: Emitido quando o **Agente Comunicador** envia um pedido ao software embarcado, ou seja, quando um elemento na fila de envio é processado.

Os *signals* **New Sample** e **Reply to a previous request** são conectados a *slots* do **Agente Interpretador** que processa o *signal* recebido e se necessário repassa os dados ao **Agente Documentador** ou ao **Módulo de Visualização**. Além disso os três *signals* são conectados a *slots* do **Módulo de Estatísticas**.

5.3 Uso de *Mock Objects*

Como o hardware e o software embarcado foram desenvolvidos em paralelo com o software supervisor, era impossível testar o funcionamento do software supervisor em conjunto com o hardware real. Para suprir esta deficiência, utilizou-se a tecnologia de *Mock Objects*[3].

Foram desenvolvidos neste trabalho dois *mock objects*:

Mock Tool: Usado para simular de modo controlado a ferramenta de inspeção.

Mock Interpreter: Usado para analisar os dados recebidos e enviados pelo **Agente Comunicador**.

5.3.1 Mock Tool

O *Mock Tool* é um simulador completo da ferramenta de inspeção, tanto do software embarcado quanto do hardware (Figura 5.3). O *Mock Tool* funciona com uma interface gráfica aliada a um mini interpretador de *scripts*. Com isso é possível tanto operar manualmente o simulador, como gerar seqüências de ocorrências ou comportamentos programados. Em ambos os casos, abre-se a possibilidade de testes de comandos em lote, de forma a verificar como as combinações de eventos podem afetar o software.

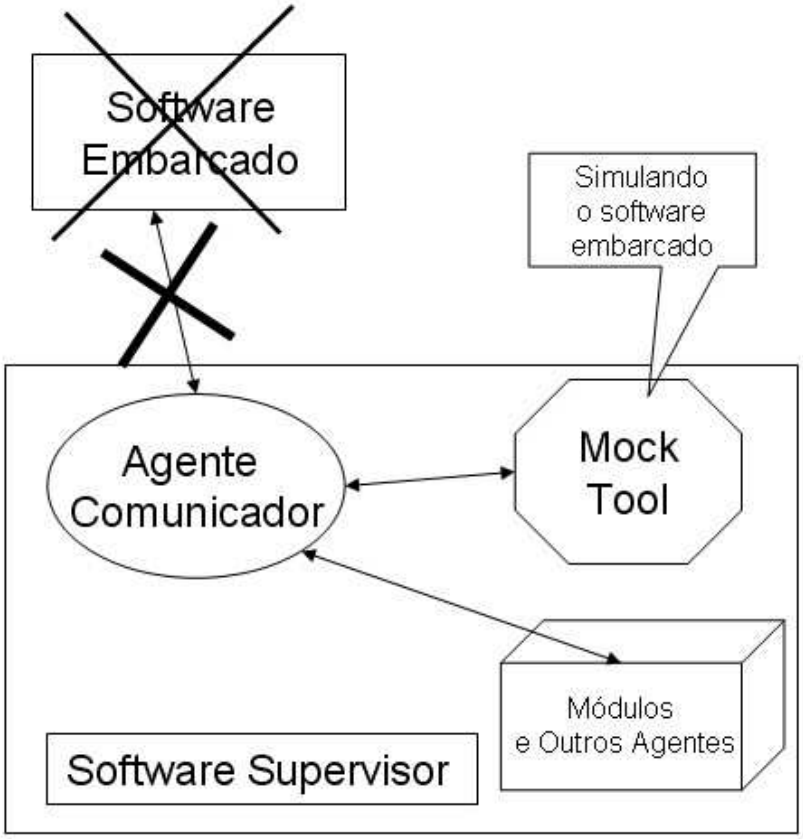

Figura 5.3: Diagrama - *Mock Tool*

Através dele é possível simular condições de execução variadas, tanto normais como anormais, tais como:

Variação brusca de velocidade na ferramenta: Variações bruscas são esperadas devido à forma manual como é a movimentação da ferramenta de inspeção sobre o duto, que é feita através da força humana, sendo os operadores responsáveis por "puxar o cabo". Nestes casos, pode ser

que a transmissão dos dados não atenda à taxa necessária para que o espaço entre duas amostras consecutivas fique dentro da precisão desejada (que pode variar, mas é da ordem de poucos milímetros). O software supervisor deve ser capaz de identificar tais casos e solicitar ao operador que inspecione novamente os pontos onde a precisão estiver aquém do desejado.

Inversão de sentido: A ferramenta de inspeção pode ir tanto para frente, quanto para trás durante uma inspeção. O software supervisor deve ser capaz de identificar este caso, mostrar este dado ao operador, e tratá-lo (por exemplo: alguns cálculos levam em conta o sentido da ferramenta).

Perda de comunicação: O software supervisor deve ser capaz de tratar os casos em que a comunicação se perde (por má qualidade de sinal, erro de hardware, etc.).

Erros na ferramenta: O software supervisor deve ser capaz de analisar criticamente alguns parâmetros recebidos da ferramenta de forma a identificar anomalias de funcionamento (tais como sensores com problemas, odômetros desregulados, bateria enfraquecida, etc.)

Falhas na comunicação: A ocorrência de dados inconsistentes ou mal formados é freqüente e deve ser tratada de forma que sejam imediatamente reconhecidos e descartados.

Um ponto relevante a ser discutido é que o *Mock Tool* faz uso do mesmo componente de comunicação usado pelo sistema supervisor, o que não só contribuiu para que esse componente fosse um componente confiável, como também demonstrou seu caráter geral, podendo ser reutilizado em outros projetos que trabalhem com *bluetooth*.

5.3.2 Mock Interpreter

O *Mock Interpreter* é um componente que é acoplado aos *signals* emitidos pelo **Agente Comunicador**. Com ele é possível verificar a confiabilidade do protocolo de transporte implementado na camada de comunicação. Isso foi feito em conjunto com uma pequena ferramenta de envio de pacotes de *bytes*: eram enviadas baterias de *bytes* conhecidos, contendo tanto pacotes válidos como inválidos com as mais variadas configurações, e os resultados das validações de consistência eram analisados. (Figura 5.4).

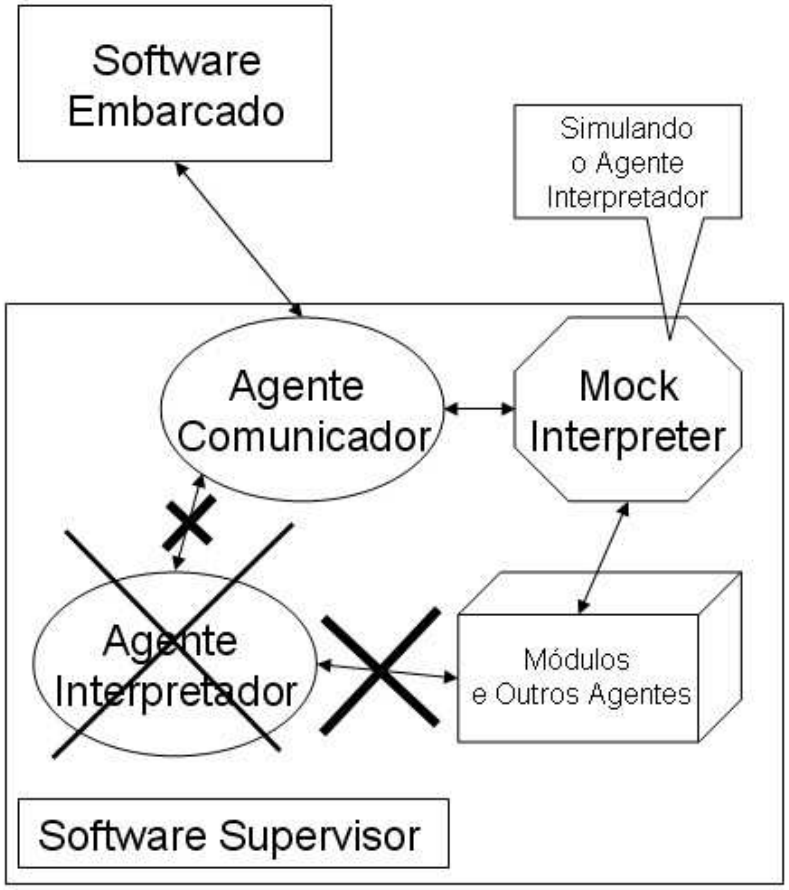

Figura 5.4: Diagrama - *Mock Interpreter*

5.4
Usando Assertivas

5.4.1
O Módulo *assert.cpp*

Foi gerado um módulo específico para o tratamento e a verificação das assertivas. Este módulo, chamado *assert.cpp*, possui um conjunto de funções e macros destinadas à utilização de assertivas. Os principais elementos deste módulo são as macros *ABORT*, *ASSERT* e *ASSERT_VALID*.

ABORT: É a macro que aborta o programa. Antes de abortar o programa ela exibe uma mensagem ao usuário e registra essa mensagem no *log* (normalmente em um arquivo). Esta mensagem contém o texto passado

como parâmetro para a macro, a versão do software que está sendo executada, e a linha e o arquivo em que a macro foi invocada.

Assinatura da macro: Código Fonte 5.1

```
1 ABORT(<mensagem>[, <argumentos>]);
```

Código Fonte 5.1: Assinatura da macro ABORT

Exemplo de uso: Código Fonte 5.2

```
659 ...
660 QFile* file = new QFile(fileStr);
661 //fileStr="C:/corridas/corrida20051111.grf"
662 if(file->open(IO_ReadOnly))
663 {
664 ...
665 }
666 else
667 {
668     ABORT(tr("O arquivo da base de dados \"%s\" não pode ser aberto."),
          fileStr);
669     //tr(const char*) é uma função do Qt que permite a internacionalização
          automática das strings do código.
670 }
671 ...
```

Código Fonte 5.2: Exemplo de uso da macro ABORT

Mensagem exibida ao usuário: Figura 5.5

ASSERT: Define uma assertiva. Se a expressão booleana passada como argumento for inválida, o programa é abortado. Como na macro *ABORT*, antes de abortar o programa ela exibe uma mensagem ao usuário e registra essa mensagem no *log* (normalmente em um arquivo). Essa mensagem contém o texto passado como parâmetro para a macro, a versão do software que está sendo executada, a expressão booleana da assertiva, e a linha e o arquivo em que a macro foi invocada.

Assinatura da macro: Código Fonte 5.3

```
1 ASSERT(<expressão boolean>)(<mensagem>[, <argumentos>]);
```

Código Fonte 5.3: Assinatura da macro ASSERT

Exemplo de uso: Código Fonte 5.4

Mensagem exibida ao usuário: Figura 5.6

ASSERT_VALID: Esta macro é uma extensão da *ASSERT* para um dos testes mais comuns, testar se um ponteiro é válido. Se o ponteiro não for válido o programa é abortado. Como na macro *ASSERT*, antes de abortar o

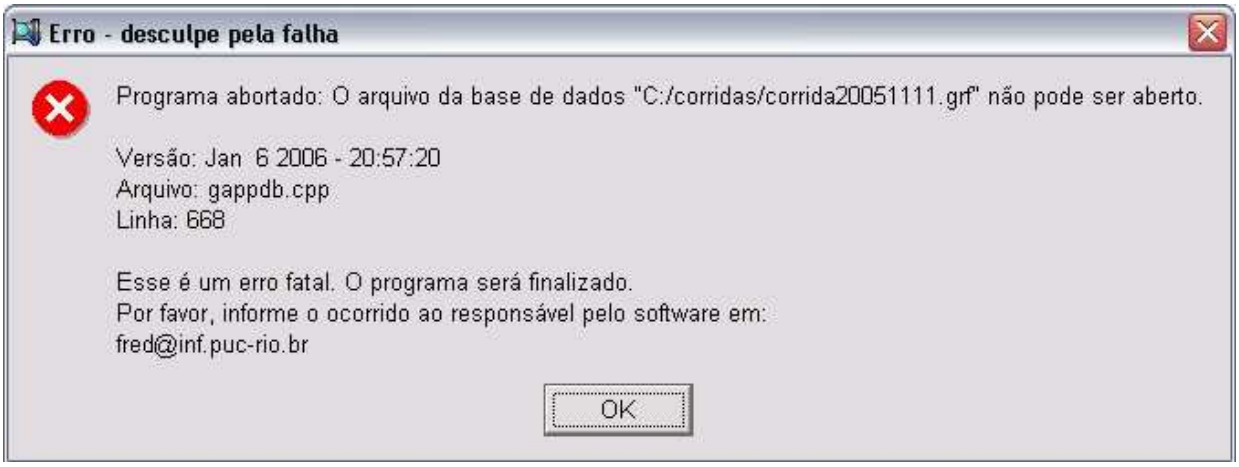

Figura 5.5: Exemplo de mensagem exibida pela macro *ABORT*


```
124 ...
125 int sensorIx = 0;
126 ...
127 ASSERT((sensorIx < MAX_SENSORS) && (sensorIx >= 0))
128     (tr("Sensor inválido (%d). Número de sensores %d"), sensor, MAX_SENSORS);
129     //tr(const char*) é uma função do Qt que permite a internacionalização
            automática das strings do código.
130 ...
```

Código Fonte 5.4: Exemplo de uso da macro ASSERT

Figura 5.6: Exemplo de mensagem exibida pela macro *ASSERT*

programa, ela exibe uma mensagem ao usuário e registra essa mensagem no *log* (normalmente em um arquivo). Esta mensagem contém o texto passado como parâmetro para a macro, a versão do software que está sendo executada, a expressão booleana da assertiva, uma explicação de porque o ponteiro é inválido, além da linha e do arquivo em que a macro foi invocada.

Para testar se um ponteiro é válido a macro faz uso de um conjunto de funções de validação de ponteiros. Foi criada uma função para validar cada grupo de ponteiros, essas funções possuem uma assinatura semelhante (Código Fonte 5.5), variando apenas o tipo de ponteiro que recebem como parâmetro. Elas recebem o ponteiro a ser testado e um ponteiro para uma *QString*, e retornam *true* se o ponteiro for válido. Caso o ponteiro seja inválido a *QString** passada como parâmetro recebe uma *string* que é uma mensagem contendo o(s) problema(s) encontrado(s) durante a validação.

```
1  ...
2  //Testa se pointer é ou não NULL, e faz validações da estrutura da lista.
3  bool isValid(SortedList* pointer, QString* problemMessagePtr);
4
5  //Testa se pointer é ou não NULL, e testa pointer utilizando seu próprio método,
       o método isValid() do AppObject.
6  bool isValid(AppObject* pointer, QString* problemMessagePtr);
7
8  //Apenas testa se pointer é ou não NULL.
9  //Essa função é invocada no caso de ponteiros que não possuem uma função
       específica para eles
10 bool isValid(void* pointer, QString* problemMessagePtr);
11 ...
```

Código Fonte 5.5: Algumas funções para validação de ponteiros

Assinatura da macro: Código Fonte 5.6

```
1  ASSERT\_VALID(<ponteiro>)(<mensagem>[, <argumentos>]);
```

Código Fonte 5.6: Assinatura da macro ASSERT_VALID

Exemplo de uso: Código Fonte 5.7

Mensagem exibida ao usuário: Figura 5.7 e Figura 5.8

5.4.2 Como Ligar e Desligar as Assertivas

Um problema que foi encontrado inicialmente no uso das assertivas (e que é comum no uso de assertivas em C/C++ e também em algumas outras linguagens), é que ou todas as assertivas estavam ligadas ou todas as assertivas estavam desligadas (quando uma assertiva está desligada ela não é testada e o

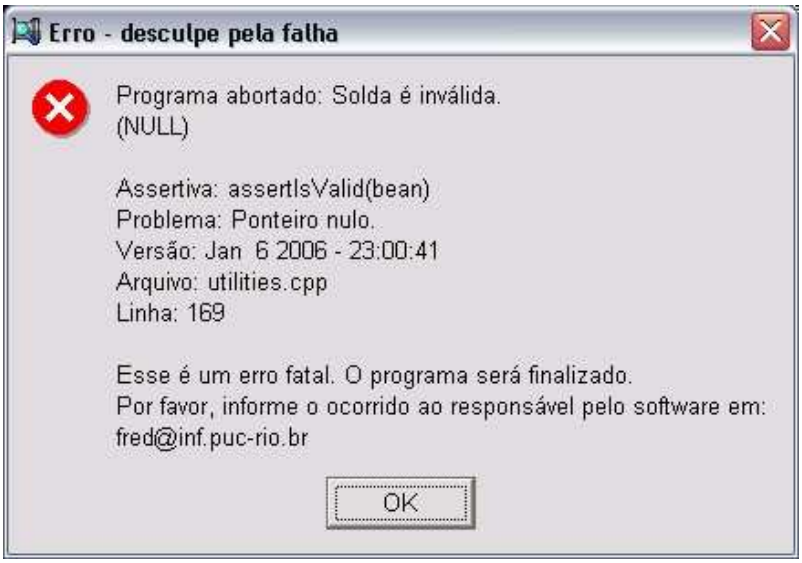

Figura 5.7: Exemplo de mensagem exibida pela macro *ASSERT_VALID*

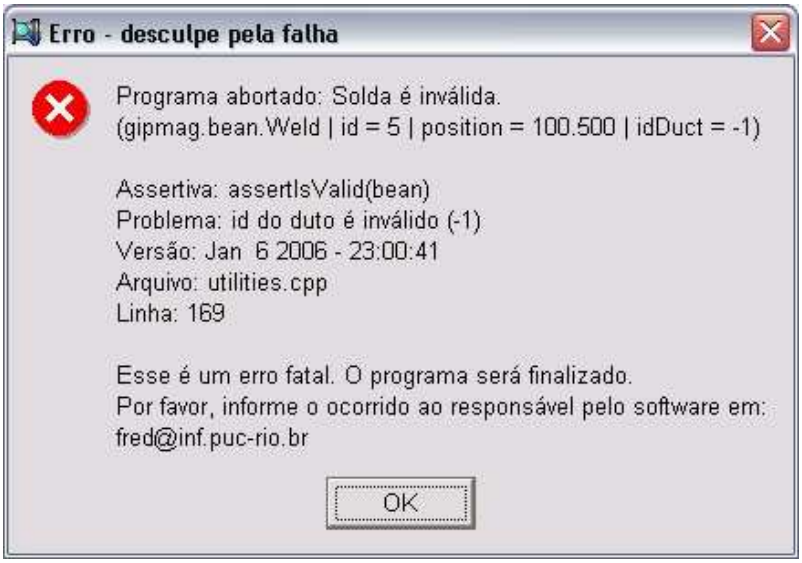

Figura 5.8: Outro exemplo de mensagem exibida pela macro *ASSERT_VALID*

```
166
167 VisWeld* Utilities::createVisualization(Weld* bean, double visPosition, int
        sample)
168 {
169     ASSERT_VALID(bean)(tr("Solda é inválida.\n%s"), toString(bean).latin1());
170     //Utililities::toString(AppObject* obj) é uma função que retorna a QString
            "NULL" se obj for NULL e obj->toString() caso contrário.
171     //tr(const char*) é uma função do Qt que permite a internacionalização
            automática das strings do código.
172     //QString::latin1() é uma função que retorna um const char* que representa a
             mesma string que o QString. Isso é necessário já que a QString não
            pode ser usada diretamente como parâmetro extra para o "%s" em um
            sprintf (e outras funções semelhantes).
173     ...
174 }
175 ...
```

Código Fonte 5.7: Exemplo de uso da macro ASSERT_VALID

programa executa como se aquela linha não existisse). Uma estratégia comum no caso de C/C++ é definir uma macro no pré-processador que habilita ou não a compilação das assertivas. Essa foi a estratégia inicialmente utilizada. Porém, isso trazia dois problemas: o primeiro é que ou todas as assertivas estavam ligadas ou todas as assertivas estavam desligadas; o segundo problema, é que uma vez compilado, ou o programa tem ou não tem as assertivas, não sendo possível ligá-las ou desligá-las sem recompilar o programa.

Durante o desenvolvimento, sentiu-se a necessidade (principalmente em relação às versões liberadas para os clientes, mesmo ainda nas fases de teste), de deixar apenas parte das assertivas ligadas e de ligar ou desligar algumas assertivas sem que fosse necessário recompilar o programa, seja porque certas assertivas eram pesadas (consumiam muitos recursos, ou demoravam muito tempo para executar) ou porque desejava-se que apenas os pontos mais críticos (ou relacionados a alguma parte específica do sistema) estivessem com as assertivas ligadas.

Uma solução encontrada para as assertivas mais pesadas (aquelas cujos testes demoram ou consomem muitos recursos) foi aproveitar a configuração do *log* do sistema (que foi escrito usando log4cxx[30]) para controlar a sua habilitação. Isso foi feito classificando-se algumas assertivas de acordo com os *Levels* do log4cxx, de tal forma que uma assertiva somente é testada se o seu respectivo *Level* estiver habilitado no *Logger* da classe (Código Fonte 5.8).

Em algum casos, o teste da assertiva tem um custo tão baixo que o simples custo da operação para testar se um determinado *Level* está ou não habilitado supera o custo do teste da assertiva. Nesses casos o *log* não foi usado em conjunto com essas assertivas (Código Fonte 5.9).

Em outros casos (principalmente nos casos de validação de consistência de ponteiros), não se deseja desabilitar a assertiva, mas apenas a parte mais pesada

```
1  ....
2  void ViewManager::openPassage(Passage* bean)
3  {
4      ...
5      QString passageFile = bean->getDataFile();
6      if(LOG->isDebugEnabled())
7      {
8          //Esse teste envolve acesso ao sistema de arquivos, que é mais custoso
               que testar se o Level DEBUG está habilitado.
9          ASSERT(QFile::exists(passageFile))(tr("O arquivo \"%s\" foi deletado ou
               movido para outra pasta."), passageFile.latin1());
10     }
11     //ou
12     //ASSERT_DEBUG(QFile::exists(passageFile))(...);
13     ...
14 ...
15 //tr(const char*) é uma função do Qt que permite a internacionalização
       automática das strings do código.
16 //QString::latin1() é uma função que retorna um const char* que representa a
       mesma string que o QString. Isso é necessário já que a QString não pode ser
        usada diretamente como parâmetro extra para o "%s" em um sprintf (e outras
        funções semelhantes).
17 ...
18 }
19 ...
```

Código Fonte 5.8: Utilizando o *log* para desabilitar assertivas - Parte I

```
1  ....
2  Comparable* SortedList::elementAt(uint16 ix)
3  {
4      //O teste dessa assertiva é menos custoso que testar se um determinado Level
           está habilitado, logo não vale a pena usar o log em conjunto com essa
           assertiva
5      ASSERT(ix < this->getSize())(tr("Índice inválido %d. List tem apenas %d
           elemento(s)"), ix , this->getSize() ), fileStr);
6      ...
7  }
8  ...
```

Código Fonte 5.9: Utilizando o *log* para desabilitar assertivas - Parte II

dos testes de validação. Nesses casos, a assertiva é usada normalmente (sem o *log*), mas o código de validação testa se um determinado *Level* está habilitado antes de fazer as verificações mais pesadas (Código Fonte 5.10).

Essa solução não elimina todo o custo de execução das assertivas, ou dos códigos de validação mais pesados, uma vez que mesmo com eles desabilitados, ainda há o custo de testar se um determinado *Level* está ou não habilitado. Mas para o estudo de caso, essa solução se mostrou satisfatória, permitindo através do *log*, que as assertivas e testes mais pesados fossem habilitados ou desabilitados de acordo com as necessidades dos programadores e/ou clientes sem que o programa precisasse ser recompilado. No entanto, existem sistemas onde mesmo o overhead de testar se um determinado *Level* está ou não habilitado

```
1  ...
2  SortedList Utilities::filterDefects(SortedList* defects, bool analysed)
3  {
4      //Essa assertiva deve sempre ser testada. Ela invoca o método
           isValid(SortedList* pointer, QString* problemMessagePtr). Já esse
           método faz mais ou faz menos testes dependendo de como o log está
           configurado.
5      ASSERT_VALID(lista)(tr("Lista é inválida."));
6      ...
7  }
8  ....
9  bool Utilities::isValid(SortedList* pointer, QString* problemMessagePtr)
10 {
11     if(pointer == NULL)
12     {
13         *problemMessagePtr = tr("Ponteito é nulo");
14         return false;
15     }
16     if(LOG->isDebugEnabled())
17     {
18         //apenas se o Level DEBUG está habilitado a estrutura da lista é
               validada
19         ...
20         //código que testa a estrutura da lista
21         ...
22     }
23     return true;
24 }
25 ...
```

Código Fonte 5.10: Utilizando o *log* para desabilitar assertivas - Parte III

não é tolerado. Nesses casos, a solução aqui apresentada não se aplica.[1]

5.4.3 Conclusões do Uso de Assertivas

O esforço extra no desenvolvimento de assertivas reduziu o esforço na fase de testes, que foram semi-automatizados. As assertivas, na maior parte dos casos, deram uma boa idéia do que estava errado (quando não mostraram exatamente a causa do erro).

A obrigação de ter que escrever assertivas durante a implementação estimulou a cautela dos desenvolvedores e, mesmo não podendo comprovar, os indícios de que este procedimento ajudou a reduzir o número de erros são extremamente fortes.

A escrita de funções de "Arrumação da Casa"[6] garante que quando uma assertiva falha, o usuário não perde seu trabalho, afinal, não é uma questão de "se o software vai falhar", mas "quando o software vai falhar". De fato, o sistema

[1] Algumas linguagens como Java[23], que possui um mecanismo próprio para o uso de assertivas (o *assert*), permitem determinar no início da execução (através de parâmetros de execução) quais assertivas estarão ligadas e quais estarão desligadas, no entanto o código final gerado a partir de tais linguagens são, em geral, mais lentos do que C/C++ o que descarta completamente o seu uso em sistemas com grandes requisitos de performance.

está sendo usado em inspeções reais de dutos. Durante essas inspeções algumas assertivas falharam, mas nenhum trabalho foi perdido.

Os resultados obtidos pelo uso de assertivas foi tão bom que, as assertivas foram mantidas ligadas (exceto as assertivas com testes muito pesados) no código da versão do sistema que está em produção.

5.4.4 O Próximo Passo: *AssertionHandler*

Uma questão observada durante o desenvolvimento foi que muitos dos pontos onde o programa era abortado (devido à identificação de falha em assertivas) poderiam ser, com algum esforço de projeto e implementação, transformados posteriormente em pontos de recuperação. Percebeu-se que a opção de se abortar o programa em caso de falha no teste nem sempre é a melhor opção, pois em alguns casos é possível que o programa se recupere daquele estado inválido ou inconsistente.

Muitos dos pontos de término forçado de execução foram transformados em código de recuperação, que analisa o estado inconsistente do sistema e realiza as correções necessárias para levar o sistema a um estado válido de execução. Muitas vezes, a correção incorre em perda de dados e, nesses casos, é preciso consultar o usuário acerca do que deve ser feito. Muitos pontos também requerem alguma intervenção do usuário, o qual deve definir alguns parâmetros de recuperação. Dessa forma, aumentou-se a capacidade de recuperação, reduzindo os casos onde seria necessário reinicializar o sistema.

O processo de transformação dos pontos onde o programa era abortado, em código de recuperação, não teve nenhum tipo de projeto prévio ou estruturação: os pontos foram identificados e o código de término forçado de execução (que fica dentro do *if* do teste de assertiva) foi trocado pelo código de recuperação. Dessa forma, o que ocorreu foi uma mistura de código de lógica funcional do sistema com código de verificação de assertivas e recuperação de falhas. Na maioria dos casos, o código de recuperação era complexo e extenso, e mesmo adotando técnicas básicas de modularização (tais como criação de métodos), a sua presença não passava despercebida nas classes onde estava presente. Este problema motivou a procura por estratégias que pudessem aumentar a qualidade de engenharia do tratamento de falhas no sistema, e inspirou a criação do chamado *AssertionHandler*.

O *AssertionHandler* é um componente cujo objetivo é concentrar todas as responsabilidades relativas à verificação e recuperação de falhas. É baseado no conceito da existência de *verificação*, *verificadores* e *tratadores*.

Condição	Verificador	Tratador
VALID_DOC	AssertVerifierValidDoc	FailureHandlerInvalidDoc
VALID_TOOL_CONFIG	AssertVerifierValidToolConfig	FailureHandlerInvalidToolConfig

Tabela 5.1: Exemplo de tabela de configuração do *AssertionHandler*

Uma *verificação* é um conjunto de validações de assertivas que devem ser feitas em um determinado ponto de execução do sistema. É uma espécie de rótulo, que marca o ponto onde um determinado estado do sistema deve ser garantido a fim de garantir a consistência do código.

Um *verificador* é um componente responsável por verificar um determinado conjunto de assertivas no sistema. Ele implementa a interface *AssertionVerifier*, numa aplicação do padrão de projeto *Command*[53]. Possui o método *verify*, cujo retorno é um booleano indicando se a condição foi ou não satisfeita.

Um *tratador* é um componente responsável por tratar a ocorrência de uma falha no sistema. Ele implementa a interface *FailureHandler*, numa aplicação do padrão de projeto *Command*[53]. Possui o método *handleFailure*, que retorna um booleano indicando se foi possível ao tratador se recuperar da falha localizada.

O diagrama de classes simplificado do *AssertionHandler* é exibido na Figura 5.9.

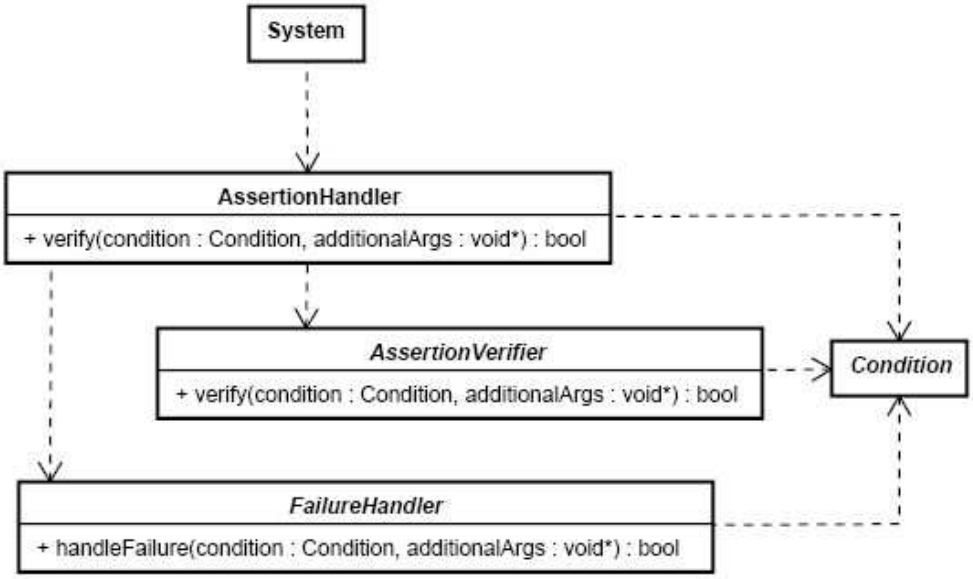

Figura 5.9: Diagrama de classes simplificado do *AssertionHandler*

O *AssertionHandler* possui uma tabela, definida em arquivo de configuração, que vincula condições, verificadores e tratadores. A estrutura da tabela será apresentada a seguir (Tabela 5.1):

Para cada condição há um verificador cadastrado, responsável por testar o conjunto de assertivas componentes da condição. Caso as assertivas falhem, há o *handler*, que deverá ser ativado a fim de tentar recuperar o sistema.

Quando o seu método *verify* é invocado, o *AssertionHandler* localiza o *AssertionVerifier* cadastrado para a condição encontrada. Caso a invocação do método *verify* (do *AssertionVerifier*) retorne *true*, o processo de verificação se encerra, uma vez que o estado do sistema é consistente com as operações que deverão ser realizadas a seguir. Porém, se o retorno for *false*, ao menos uma assertiva falhou. No segundo caso, o *AssertionHandler* localiza o tratador responsável por aquela condição e invoca o seu método *handleFailure*. Este método deverá executar rotinas de "Arrumação da Casa"[6], a fim de tentar fazer com que o sistema atinja um estado consistente. Um retorno *true* indica que a recuperação foi possível, caso em que o sistema pode seguir sua execução normalmente. Porém, se o retorno for *false*, não há nada mais a fazer, a não ser encerrar a execução do sistema (Figura 5.10).

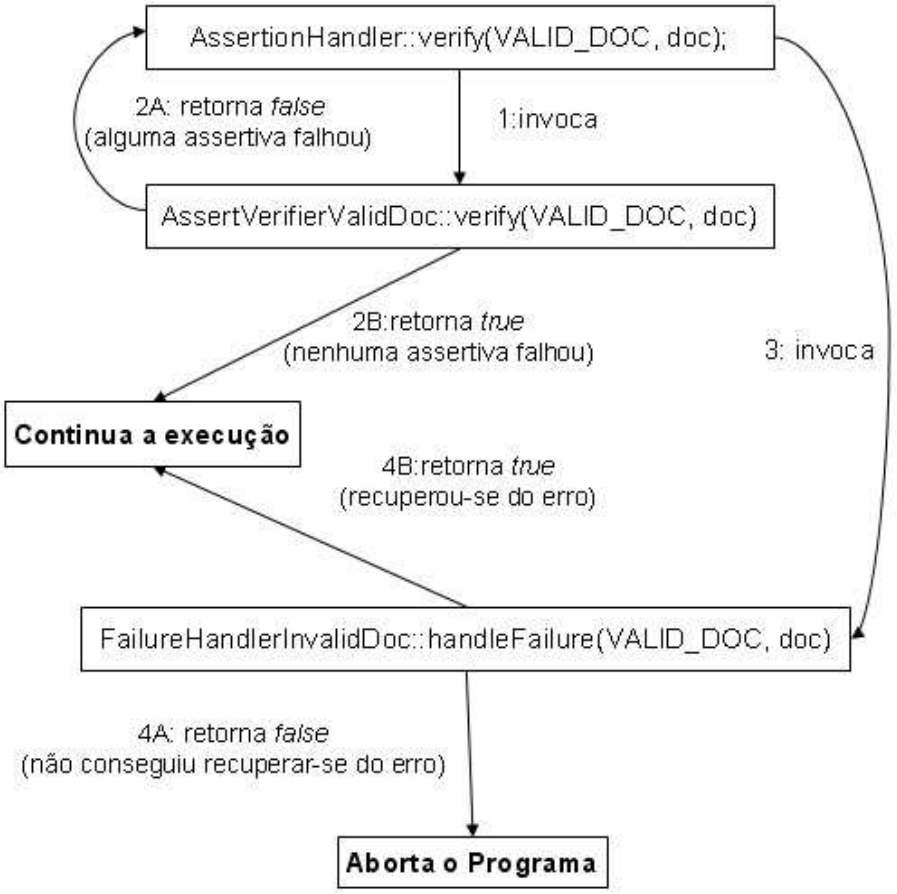

Figura 5.10: Fluxo de execução do *AssertionHandler*

Conseqüências do uso do *AssertionHandler*

O *AssertionHandler* possibilita a separação do código de validação e recuperação do código funcional do sistema, o que, a primeira vista, promove uma maior organização. Além disso, permite que o desenvolvimento seja paralelizado, onde uma equipe fica a cargo dos verificadores e tratadores enquanto a outra é responsável pelo código funcional propriamente dito. Permite ainda o rápido desligamento da verificação de assertivas, o que pode ser feito com a substituição do *AssertionHandler* por um objeto falso, que retorne *true* no método *verify*. Outra possibilidade é a de criar um *AssertionVerifier* que sempre retorne *true* em seu método *verify*, alternativa que permite um maior controle sobre quais condições devem ser verificadas durante a execução. Como toda a parte de configuração está separada em um arquivo, é possível criar diferentes níveis de verificação e mantê-los arquivados para utilização em diferentes situações como: testes de homologação, ambiente de produção, testes de um subsistema específico, etc..

Há casos, no entanto, em que uma recuperação não é viável e a melhor opção é abortar a execução do sistema. Isso é modelado através da criação de um *FailureHandler* que retorne *false* em seu método *handleFailure*.

Um ponto negativo dessa abordagem é que, ao mesmo tempo em que a separação do código de verificação e tratamento do código funcional promove uma maior organização, ela diminui a chamada modularização de raciocínio, pois para a completa compreensão de um trecho de código, passa a ser necessário consultar diversos elementos que estão em pontos separados.

5.5 *Design by Contract* e *Mock Elements*

O software supervisor foi desenvolvido com um objetivo claro: o número de faltas presentes no software deveria ser o menor possível. Para tanto, desde os primeiros momentos do desenvolvimento houve uma forte preocupação com a qualidade final dos artefatos gerados em todos os níveis de abstração. Métodos, classes e subsistemas foram gerados com um cuidado especial, o de que a identificação de falhas ocorresse da forma mais automatizada possível. Com isso, esperava-se não só aumentar a qualidade final, mas também facilitar o processo de depuração e, portanto, reduzir a fase de testes e homologação.

Durante o desenvolvimento do sistema, o uso das técnicas pregadas pelo DBC foi uma constante. Todos os métodos de maior complexidade possuem pré-condições. Muitos deles tiveram suas pós-condições escritas. As

pré e pós-condições foram transformadas em assertivas executáveis, as quais foram convertidas em código inserido diretamente nos métodos. As assertivas executáveis foram implementadas de tal forma que, ao serem ativadas, abortam imediatamente a execução do software e exibem uma mensagem de erro contendo uma identificação acerca da falha ocorrida, além de valores úteis ao processo de depuração (estado de variáveis locais, atributos da classe, localização do erro, etc.).

Embora seja impossível afirmar, são grandes as evidências de que o esforço extra durante o desenvolvimento gerado pela redação das pré e pós-condições e pela implementação das assertivas executáveis foi o grande responsável não só pelo reduzido tempo de teste do sistema, mas também pelo reduzido tempo médio de depuração das falhas encontradas. O número de falhas identificadas também pode ser considerado pequeno, o que pode ser visto no Capítulo 6. Uma possível razão para tanto é o fato de que a obrigação na redação de pré e pós-condições força o programador a pensar minuciosamente sobre o seu trabalho, o que acaba por gerar um código mais propenso a estar correto.

Um ponto interessante a ser discutido foi que, durante boa parte do desenvolvimento, não se tinha acesso ao protótipo do hardware do sistema, pois este ainda estava em desenvolvimento. Para suprir essa necessidade, foi desenvolvido um simulador cujo objetivo era o de fornecer dados controlados para o teste do software supervisor. Esta iniciativa teve como base teórica a idéia dos *Mock Objects*[3], com a diferença que, neste caso, o elemento falso gerado mais se assemelhava a um *Mock Component*, ou um *Mock Agent*, dada a sua natureza independente (do sistema), pró-ativa e altamente configurável. Essa estratégia se mostrou tão bem-sucedida que, quando o sistema foi testado com o protótipo real, ocorreram apenas poucos erros - todos imediatamente capturados pelas assertivas executáveis - que foram corrigidos rapidamente, de forma que em menos de quatro horas havia um sistema completo, desde o hardware do equipamento até o software supervisor.

A versão do software inicialmente liberada para o ambiente de produção foi compilada com as assertivas executáveis ativas. Esta versão foi inicialmente utilizada em ambiente controlado, de desenvolvimento, cujo objetivo era permitir o estudo e o aprimoramento da eletrônica utilizada, tanto em nível de confiabilidade (resistência às intempéries do ambiente como chuvas, quedas, etc.) quanto em nível de acurácea na aquisição dos dados (ajustes dos sensores). Neste ambiente ainda foram identificadas e corrigidas algumas anomalias, todas capturadas por assertivas executáveis. Em um segundo momento, liberou-se a versão para o ambiente de produção, também com as assertivas executáveis ativas. A quantidade de faltas apresentada foi muito pequena: em dois meses de

uso, apenas em dois momentos o software falhou, sempre caindo em assertivas. O procedimento de relatório de anomalias instituído foi o envio do arquivo de *log* do sistema, anexando um pequeno comentário acerca do que estava sendo feito no momento da falha e, se possível, um *printscreen* da tela no momento do erro (ou logo antes do erro), no caso de um erro que pode ser reproduzido facilmente. Com isso, era possível depurar o sistema de uma forma mais eficiente, pois havia uma grande quantidade de informação disponível. Ainda não foi liberada uma versão do software com as assertivas executáveis desativadas, mas os resultados obtidos em termos de melhorias no desenvolvimento em detrimento a uma pequena redução na performance vêem sendo tão positivos que não há sequer uma previsão de liberação de uma versão sem as assertivas.

Desde que foi liberada para produção, a versão inicial sofreu diversas intervenções para a criação de novas funcionalidades. As assertivas executáveis continuaram ajudando, haja vista que minimizam o impacto de anomalias existentes devido à alteração de interfaces e/ou de incompatibilidade de comportamento entre os componentes do sistema.

A presença de assertivas em todo o código (inclusive de assertivas básicas como testar se uma variável é nula ou tem um valor inconsistente) tornou mais rápida a descoberta e o conserto de erros de programação, mesmo quando os erros ocorreram enquanto o sistema era utilizado em um ambiente real.

O uso de assertivas para evitar erros de acesso (acessar uma variável com valor *NULL* ou não inicializada, ou um acesso inválido de memória) se mostrou indispensável principalmente devido à linguagem usada no desenvolvimento - em C/C++ um erro dessa natureza faz o programa encerrar sua execução por *Falha de Segmentação*, sem que se tenha qualquer informação sobre o que aconteceu ou onde aconteceu[2]. Em linguagens mais recentes (como Java[23]), esse tipo de assertiva não é tão importante, uma vez que um erro dessa natureza, ao invés de fazer o programa abortar sem maiores informações acerca da falha ocorrida, levanta uma exceção que permite o conhecimento exato do que aconteceu, e em qual linha de código ocorreu a falha.

[2]Em C++ é possível capturar a ocorrência de erros como *Falha de Segmentação*, e outros erros que abortam automaticamente o programa, no entanto não é possível saber exatamente o que aconteceu nem aonde aconteceu, a única informação disponível é um código que identifica de modo geral o erro ocorrido (cada sistema operacional tem uma lista de erros e do código associado a eles). Como não é possível saber aonde ocorreu o erro, e conseqüentemente em que ponto da execução o programa está, não é seguro continuar a execução do programa após interceptar um erro dessa natureza, por isso o procedimento normal nessa situação acaba sendo abortar o programa. No entanto capturar estes erros permite que se execute algum código antes que o programa seja abortado, como: liberar recursos como arquivos e conexões com banco de dados; salvar dados que o usuário estava editando; etc.

5.6 Banco de Dados

Na implementação, procurou-se isolar a camada de negócio da camada de persistência. Para isso, toda parte de acesso ao banco foi implementada em uma única classe, a *AppDB* (*ApplicationDataBase*), desenvolvida de modo a fornecer uma interface com serviços de alto nível encapsulando por completo tudo relacionado a comandos SQL e a forma como é feita a comunicação com o banco.

Os métodos da *AppDB* manipulam diretamente os objetos de dados que são persistidos, todos são sub-classes da classe *Bean*[3] (mais detalhes na próxima seção - 5.6.1). As funções permitem ações como salvar/atualizar/remover um objeto específico, obter todos os objetos de uma classe específica ou obter os objetos de uma determinada classe que tenham determinados valores em determinados atributos. Por fim, mesmo não sendo recomendado, há suporte para a execução de comandos SQL diretamente (através de funções específicas), caso seja necessário.

Internamente apenas um método, o *executeSql*, faz contato direto com o banco. Este passa uma expressão SQL para o banco e trata o retorno. As demais funções apenas montam as consultas SQLs e, em caso de um *get*, determinam quais funções auxiliares serão acionadas, a fim de processar o resultado da consulta.

O tratamento de erros no acesso à base de dados foi bastante simplificado: no momento da abertura da base, ocorre uma validação completa de sua estrutura e dos seus dados. Caso seja identificada alguma anomalia ou inconsistência, é executado um código de recuperação, que leva a base a um estado consistente (esse código pode requerer auxílio do usuário). Uma vez que a base está aberta, todo acesso é monitorado e, caso alguma falha ocorra, a execução do software supervisor é abortada com a exibição de uma mensagem explicativa da natureza do erro. Abortar a execução neste caso não afeta em nada o software embarcado, uma vez que apenas software supervisor faz uso do banco de dados; e também não há o risco de interromper as inspeções, pois o banco de dados é acessado apenas durante a análise e durante o gerenciamento da inspeção (por exemplo: para verificar que partes do duto ainda não foram inspecionadas), ele não é acessado enquanto a ferramenta de inspeção está sendo usada (que é quando o software supervisor e o software embarcado estão se comunicado). A verificação do sucesso no acesso à base é feita através do parâmetro de retorno das funções que executam o acesso. Todas as funções retornam uma *string*, que é vazia

[3] A classe *Bean* e suas sub-classes possuem uma implementação semelhante ao *javabean*[54], ou seja, são objetos que apenas possuem atributos (e os respectivos métodos de *get* e *set*).

se o acesso ocorreu sem problemas, ou é uma mensagem de erro caso algum erro tenha ocorrido. Desta forma, foi possível uniformizar a verificação através da criação de uma única função que encapsula as invocações às funções de acesso, interceptando o valor retornado e verificando se este é realmente vazio ou se houve um erro, caso em que a execução do sistema é abortada e o erro é registrado no *log* da aplicação.

5.6.1 *Beans*

Os *beans* são as classes que representam o modelo do sistema. Cada *bean* concreto representa uma das entidades do modelo, enquanto cada *bean* abstrato serve para concentrar atributos e funções comuns a alguns *beans*. Todos os *beans* concretos são persistidos na bases de dados do sistema.

Para facilitar a utilização dos *beans* pelas outras classes, cada *bean* possui um atributo que é o nome de sua classe, assim, qualquer função que receba um *bean* genérico (ou alguma de suas sub-sclasses abstratas), pode saber qual é a classe real do *bean* e internamente tratá-lo da maneira apropriada. Esta abordagem permitiu, por exemplo, que só existisse uma função para salvar os dados de qualquer *bean* na base de dados, essa função recebe um *bean* genérico e a partir do atributo, ela sabe qual é a classe real do *bean* e pode salvar os dados corretamente.

A seguir serão descritos os *beans* desenvolvidos. O Diagrama de classes dos *beans* pode ser visto na Figura 5.11

AppObject: Classe abstrata que é super-classe da maior parte das classes do sistema. Esta classe possui quatro métodos que devem ser implementados pelas suas sub-classes:

toString(): Retorna um *QString* representando o objeto. Essa *string* é usada para fins de depuração, e deve conter o estado interno do objeto. Classes que precisam ser apresentadas como uma *string* (classes que são usadas como valor em uma tabela, por exemplo) devem criar métodos próprios para isso, ao invés de usar o método *toString()*. Este método é abstrato.

getClass(): Retorna um *QString* que é o nome da própria classe. Convencionou-se que todas as sub-classes de *AppObject* devem possuir uma constante pública e estática chamada *CLASS* cujo valor deve ser igual ao valor retornado pelo método *getClass()*. Essas constantes são usadas juntamente com o método *getClass()* para saber qual é a classe

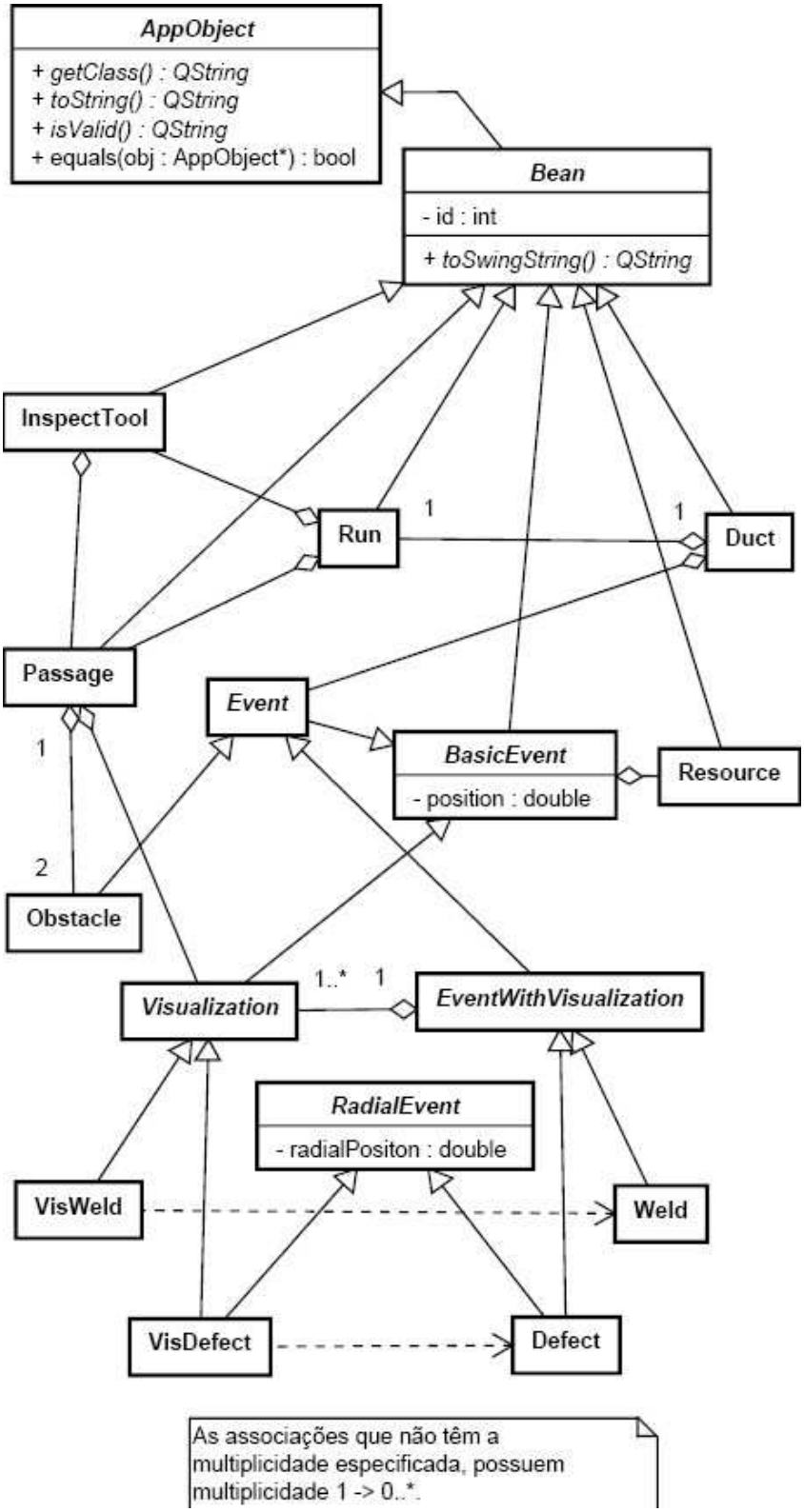

Figura 5.11: Diagrama de classes dos *Beans*

real dos objetos em funções que recebem como parâmetro um *AppObject* ou alguma de suas sub-classes abstratas. Este método é abstrato.

isValid(): Método que testa se o próprio objeto é válido (se sua estrutura está correta, se o valor de seus atributos estão em um intervalo aceitável, etc.). Este método retorna uma *QString* vazia, se o objeto estiver válido, ou uma *QString* que descreve o(s) problema(s) encontrado(s) durante a validação do objeto se ele estiver inválido. Este método é abstrato.

equals(AppObject*): Usada para testar se dois *AppObjects* são iguais. Retorna *true* se o objeto passado como parâmetro for igual ao próprio objeto. Apesar de não ser abstrato recomenda-se que este método seja sobrescrito pelas sub-classes.

***Bean*:** Classe abstrata que é super-classe de todos os *beans*. Os *beans* são identificados por um *id* único. Dois *beans* de classes diferentes podem ter o mesmo *id*, mas dois *beans* da mesma classe que tem o mesmo *id* representam o mesmo objeto. Essa classe possui um método que deve ser implementado pelas suas sub-classes:

toSwingString(): Retorna um *QString* representando o objeto. Essa *string* é usada para fins de apresentação do objeto (quando ele é usado como valor em uma tabela, ou aparece em uma mensagem do sistema). Este método é abstrato.

***Duct* (Duto):** Representa o duto a ser inspecionado.

***Run* (Corrida):** Representa uma corrida (inspeção) em um duto[4]. Uma corrida é composta de diversas passagens onde cada passagem representa a inspeção de um trecho do duto. Como pode ser visto no diagrama de classes (Figura 5.11), cada duto tem apenas uma corrida, pois, na prática, cada inspeção se restringe a um único duto. Apesar desta restrição, o banco e a implementação do duto permitem que um duto possua diversas corridas.

***InspectionTool* (Ferramenta de Inspeção):** Representa a ferramenta de inspeção. Cada ferramenta tem um alcance horário que determina a área do duto que a ferramenta cobre. Por exemplo, uma ferramenta de 12h

[4]Quando a inspeção de um duto é realizada, normalmente apenas uma parte do duto (do quilômetro 110 até o quilômetro 150 do duto, por exemplo) é inspecionada, e não toda a sua extensão. No entanto, para o sistema, o duto corresponde apenas a essa parte que será inspecionada.

consegue inspecionar toda a extensão radial do duto de uma vez, enquanto que uma ferramenta de 9h cobrirá apenas 75% do duto (Figura 5.12).[5]

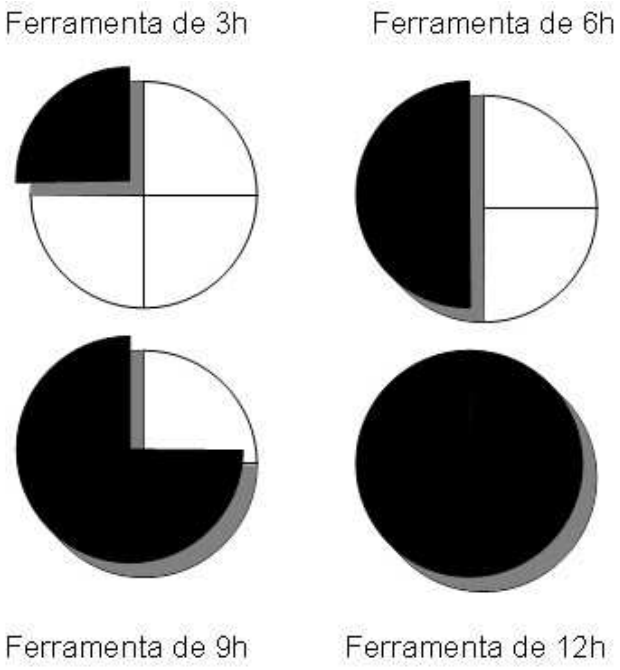

Figura 5.12: Alcance horário das ferramentas de inspeção

***Passage* (Passagem):** Uma passagem representa a inspeção de um trecho do duto entre dois obstáculos. Cada passagem está relacionada à ferramenta de inspeção usada na mesma. A partir dos obstáculos (final e inicial), e do alcance horário da ferramenta de inspeção, sabe-se exatamente a área do duto coberta por uma determinada passagem.

***BasicEvent*:** Representa qualquer elemento que possui uma posição axial. É a super-classe dos eventos e visualizações. Esta classe é abstrata.

***Resource* (Recurso):** Representa recursos (arquivos: fotos, vídeos, arquivos de texto, etc.) associados a algum elemento (visualização ou evento).

***Event* (Evento):** Representa qualquer elemento que possa ser encontrado em um duto. Um evento está relacionado diretamente ao duto. Esta classe é abstrata.

***Visualization* (Visualização):** Representa um evento observado em uma determinada passagem. Duas passagens podem cobrir uma mesma área,

[5]O alcance horário de uma ferramenta é baseado no raio do duto a ser inspecionado. Uma mesma ferramenta usada em várias inspeções pode ser considerada de 12h em uma inspeção e de 6h em outra inspeção realizada em um duto maior.

mas apresentar dados diferentes para o mesmo evento (por causa de calibração da ferramenta, pela ferramenta de inspeção ter sido diferente em cada uma das passagens, além de diversos outros fatores externos). Esta classe é abstrata.

***Obstacle* (Obstáculo):** Representa elementos (obstáculos) que impedem a passagem da ferramenta (válvulas, suportes, curvas acentuadas, pedras, etc.).

***EventWithVisualization* (Evento com Visualização):** Representa eventos que possuem visualizações. Apenas soldas e anomalias possuem visualizações. Obstáculos não possuem visualizações, pois uma vez que o obstáculo, por definição, impede a passagem da ferramenta de inspeção, a área coberta por uma passagem nunca contém um obstáculo. Esta classe é abstrata.

***RadialEvent* (Evento Radial):** Representa um elemento (evento ou visualização) que possui uma posição horária. Apenas anomalias e suas visualizações possuem posição horária. Soldas e obstáculos não possuem posição horária pois ocupam toda a extensão radial do duto.[6] Esta classe é abstrata.

***Weld* (Solda):** Representa uma solda existente no duto.

***VisWeld* (Visualização de Solda):** Representa a visualização de uma solda observada em uma passagem.

***Defect* (Defeito):** Representa uma anomalia existente no duto (desgaste, ranhuras, amassados, etc.).

***VisDefect* (Visualização de Defeito):** Representa a visualização de uma anomalia observada em uma passagem.

[6]Os obstáculos nem sempre ocupam toda extensão radial do duto, mas como impedem a passagem da ferramenta, são tratados como se ocupassem. Vale ressaltar que em alguns casos, um obstáculo pequeno pode permitir a passagem de uma ferramenta de pequeno alcance horário (quando esta é utilizada no lado oposto ao obstáculo), mas por ser uma situação bastante rara, resolveu-se não tratar este caso na implementação do sistema. No entanto, o sistema permite a realização de uma passagem entre dois obstáculos não adjacentes, o que permite realizar uma passagem onde este pequeno obstáculo seja ignorado.

5.7 Arquitetura das Janelas de Visualização

O software supervisor é uma aplicação *desktop* que consiste em uma janela principal, que possui as funções básicas do sistema. A partir da janela principal é possível abrir outras janelas que possuem funções variadas (apresentar os dados lidos pelo sistema embarcado na forma de sinais, apresentar uma tabela com todos os defeitos encontrados, etc.). Apesar da diferença entre as janelas, elas possuem algumas funcionalidades comuns, e algumas partes do sistema precisam lidar com diversas janelas diferentes. Por este motivo, foi criada uma hierarquia de classes para modelar o comportamento comum, de forma a ser possível promover a reutilização de código

Para facilitar a implementação dessas janelas foram criadas duas classes: *View* e *ViewManager*. A classe *View* é a super-classe de todas as janelas. A *View* estende de *QWidget*, que é o componente gráfico básico do Qt, e possui as funções básicas comuns a todas as janelas. A classe *ViewManager* é responsável por gerenciar as janelas, criá-las, destruí-las, e cuidar dos eventos levantados por elas.

Uma outra questão que surgiu durante o desenvolvimento foi que várias janelas diferentes podiam exibir e alterar dados de um mesmo objeto (não o mesmo objeto em memória, mas o mesmo objeto do banco de dados[7]) ou que ações em uma janela podiam ter influência em outras janelas. Parar coordenar essas alterações foi criado na *View* o método *adapToNewConfig*. Sempre que uma janela altera algum objeto ou faz alguma ação que pode ter influência em outras janelas, ela emite um sinal contendo três valores: um código que identifica a alteração ou a ação realizada; a própria *View* que emitiu o sinal; e um parâmetro extra quando necessário (por exemplo: o objeto que foi alterado). Este sinal é capturado pelo *ViewManager*. O *ViewManager* então invoca o método *adapToNewConfig* em cada uma das *Views*, passando como parâmetros os valores passados no sinal (o código da alteração, a *View* que emitiu o sinal e o parâmetro adicional). Cada *View* pode então tentar se adaptar a alteração ocorrida. No entanto em alguns casos, uma janela pode deixar de fazer sentido

[7]Para facilitar o gerenciamento da memória, as janelas não usam o mesmo objeto em memória. Se as janelas compartilhassem o mesmo objeto em memória, ao fechar uma janela seria preciso verificar se alguma outra janela aberta estava usando o objeto antes de excluí-lo da memória, o que implicaria em criar (e gerenciar) um contador de referências em cada objeto que uma janela pudesse usar. Optamos pela abordagem de que cada janela tem seus próprios objetos em memória, assim, quando uma janela se fecha, ela pode excluir da memória todos os objetos que possui sem se preocupar com as outras janelas. Deve-se observar que a abordagem de usar o mesmo objeto em memória, não eliminaria a necessidade de avisar as outras janelas que o objeto foi alterado, uma vez que as outras janelas precisariam ao menos fazer uma atualização dos dados apresentados.

ou ficar em um estado inconsistente depois que um determinado evento ocorre. Por exemplo: uma janela que exibe as visualizações de uma solda deixa de fazer sentido quando a solda em questão é removida do banco de dados. Para tratar este tipo de situação o *adapToNewConfig* tem um valor de retorno (booleano) que indica ao *ViewManager* se a *View* conseguiu ou não se adaptar às alterações ocorridas. Quando a *View* não consegue se adaptar as alterações, o próprio *ViewManager* se encarrega de fechar a *View* e avisar ao usuário que a janela foi fechada para evitar inconsistências.

As *Views* se dividem em dois tipos básicos:

***Views* de tabelas:** Exibem os dados na forma de uma tabela.

***Views* de sinais:** Exibem uma representação gráfica dos dados (desenhos, gráficos)

Um diagrama de classes resumido das *Views* (com ênfase nas *Views* de tabelas) pode ser visto na Figura 5.13.

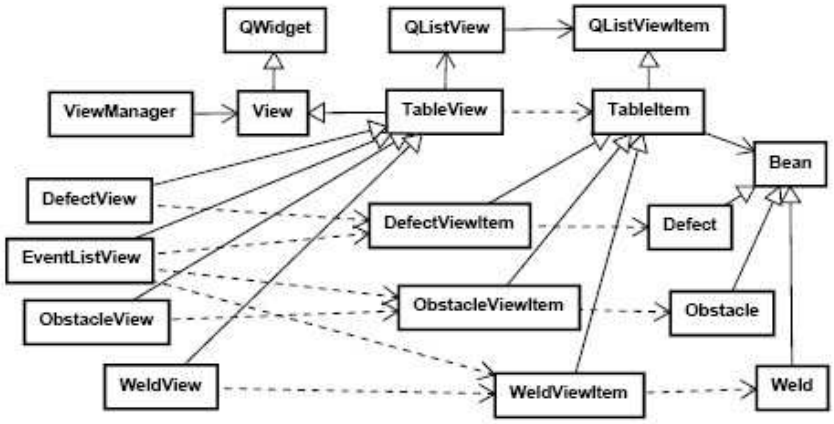

Figura 5.13: Diagrama de classes das *Views*

As seções seguintes descrevem os dois tipos de *Views*.

5.7.1 Arquitetura das *Views* de Tabelas

Um dos requisitos do sistema é permitir a visualização em tabelas dos dados dos objetos persistidos. No entanto, além de existirem diversas tabelas, sendo que algumas delas exibem dados de dois ou mais tipos de objetos ao mesmo tempo, o sistema deve permitir a criação de novas tabelas e a alteração de tabelas antigas (reordenar, remover ou adicionar colunas) sem grandes alterações no código.

As tabelas foram criadas com base na classe *QListView* do Qt, que é uma das formas de criar tabelas via Qt. Esta classe exibe instâncias da classe *QListViewItem*, onde cada linha corresponde a uma instância. A melhor forma de criar tabelas é criar novas classes herdadas da classe *QListViewItem* para modelar os itens da tabela, que são adicionados ao *QListView*. Como não seria uma boa prática fundir a hierarquia de classes de dados (oriunda da classe *Bean*) com a classe *QListViewItem*, foi adotada uma solução arquitetural que não misturasse as duas coisas. Essa solução se baseia em dividir as responsabilidades em dois grupos distintos de classes:

TableItem: Um *TableItem* estende de *QListViewItem* e é responsável por exibir e manipular objetos de uma classe *Bean* específica. Ele conhece os dados do objeto que devem ser exibidos em cada coluna (cada uma das colunas que podem existir em uma tabela possui um código identificador), conhece quais ações são possíveis sobre aquele objeto (editar, remover, adicionar recurso, etc.), além de conhecer como executar essas ações para aquele tipo de objeto específico. Existe uma, e apenas uma, classe *TableItem* para cada *Bean* existente no sistema.

TableView: Uma *TableView* estende da *View* e possui uma *QListView*. Ela define quais colunas irão aparecer na tabela e quais tipos de objetos serão visualizados na tabela. Quando uma tabela é aberta, ela busca no banco de dados os objetos que serão exibidos, e cria para cada um deles um *TableItem,* que será responsável por exibir os dados daquele objeto e executar as requisições (editar, remover, etc.) realizadas sobre o objeto. Cada *TableView* define uma tabela, assim, existe um *TableView* para cada tabela.

O primeiro passo da implementação foi criar um *TableItem* para cada *Bean*. Em seguida foram criadas as *TableViews*, uma para cada tabela a ser inserida no software supervisor. Para facilitar a criação das *TableViews* foi criada uma super-classe comum a todas elas, a *AbstractTableView* que engloba as funcionalidades

comuns a todas as *TableViews* (como por exemplo a manipulação do *TableItem*). A *TableView* faz extenso uso dos padrões de projeto *Factory Method*[53] e *Template Method*[53].

A abordagem utilizada facilita a criação de novas tabelas, além de permitir a alteração das tabelas existentes sem a necessidade de alterar várias classes, a não ser a própria *TableView* responsável por aquela tabela.

5.7.2 Arquitetura das *Views* de Sinais

As *View* de sinais são usadas quando deseja-se criar uma janela que exibirá uma representação gráfica dos dados como, por exemplo, um esboço de um defeito, um padrão de cores dependendo dos valores lidos dos sensores no sistema embarcado, etc.

Ao contrário da *View* de tabela, essas *Views* se parecem com a janela principal, podendo ter barras de ferramentas e barras de status, que permitem ações e mostram informações relacionadas aos dados exibidos.

Uma outra diferença em relação às *Views* de tabelas é o tratamento de eventos. Enquanto nas *Views* de tabelas o que importa para um evento são as linhas selecionas, nas *Views* de sinais o que importa é a posição do mouse ou a área selecionada ao se arrastar o mouse.