6
Estudos de Casos

Este capítulo apresenta diferentes estudos de caso de frameworks que foram projetados e implementados usando a abordagem orientada a aspectos proposta na tese. Para cada um dos frameworks são apresentados: (i) seu núcleo; (ii) seus pontos de junção de extensão (EJPs); (iii) seus aspectos de variabilidade e integração; e (iv) o modelo generativo OA de cada um deles.

6.1.
Framework Measurement

Measurement [17] é um framework orientado a objetos que endereça a automação de processos de medições e controle de qualidade de produtos em sistemas de manufatura. Um dos objetivos do framework é categorizar um conjunto de produtos que estão sob análise, de forma a classificá-los em aceitáveis de acordo com critérios de qualidade bem definidos. O framework *Measurement* foi implementado como parte de um estudo sistemático [78] de avaliação do uso da tecnologia de aspectos na composição entre frameworks OO. Seu projeto e implementação foi baseado em documentação publicada anteriormente [17].

6.1.1.
Núcleo do Framework

O processo de controle de qualidade do framework *Measurement* define um ciclo de análise de um item de produto. Esse ciclo de análise representa a funcionalidade principal do núcleo do framework. A Figura 26 apresenta as principais classes do núcleo do framework *Measurement*. As classes que representam pontos de extensão (*hot-spots*) são destacadas. A seguir as classes do framework são relacionadas com a funcionalidade definida para os ciclos de análise.

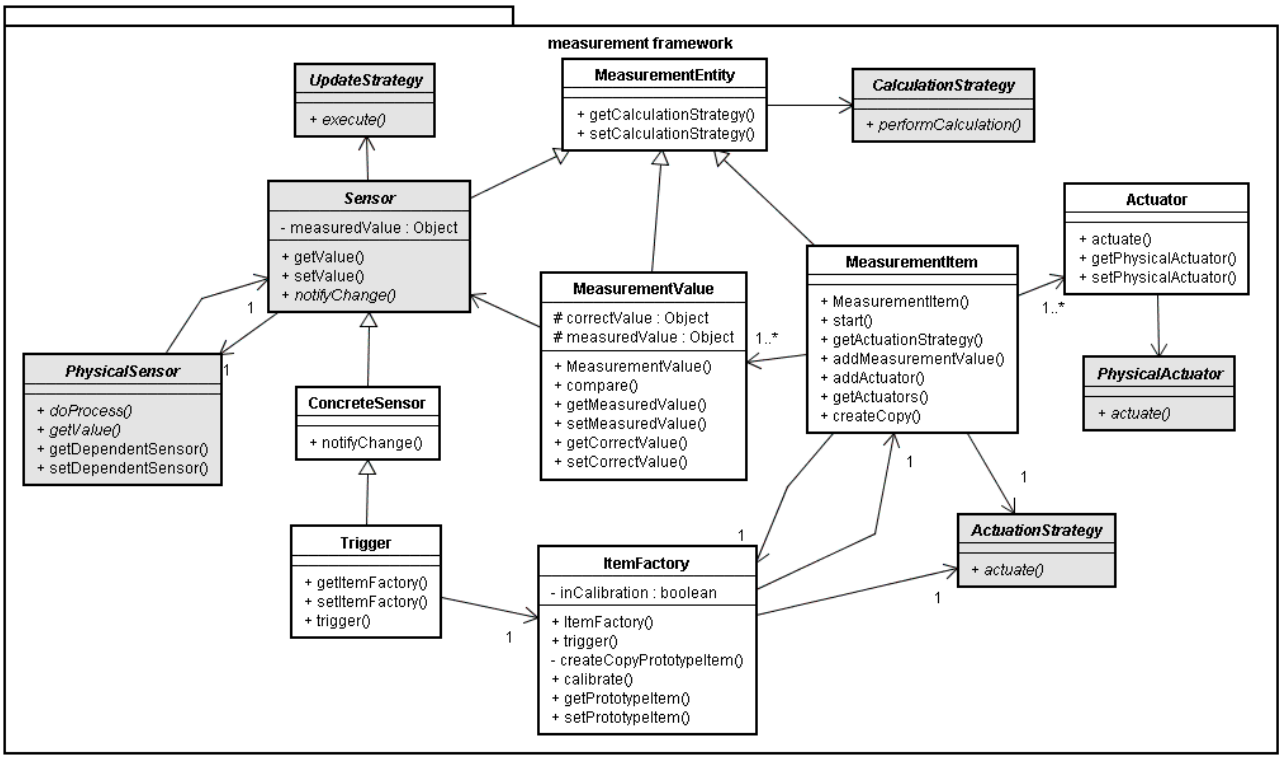

Figura 26. Diagrama de Classes do Framework Measurement

Cada ciclo de análise de um item de produto é composto pelos seguintes passos: (i) etapa de coleção de dados – um *Trigger* (classe `Trigger`) determina que um dado item de produto está entrando no sistema de medição, e a partir daí, sensores (classes `PhysicalSensor`, `Sensor` e `UpdateStrategy`) coletam propriedades relevantes do item de produto sendo avaliado; (ii) etapa de análise – os dados coletados (classe `MeasurementValue`) pelos sensores são então convertidos para uma representação comum e, em seguida, são comparados com valores ideais esperados (através das classes `MeasurementItem` e `CalculationStrategy`). Baseado nessa comparação, os itens medidos são então classificados; (iii) etapa de ação – nessa etapa ações são tomadas de acordo com a classificação dos itens (classes `Actuator`, `PhysicalActuator` e `ActuationStrategy`). Diversos tipos de ações podem ser executadas, tais como, tirar o item da linha de produção ou emitir um rótulo sobre sua qualidade. A Figura 27 mostra um diagrama de seqüência UML, ilustrando as colaborações entre as classes do framework para endereçar a implementação de um ciclo de medição.

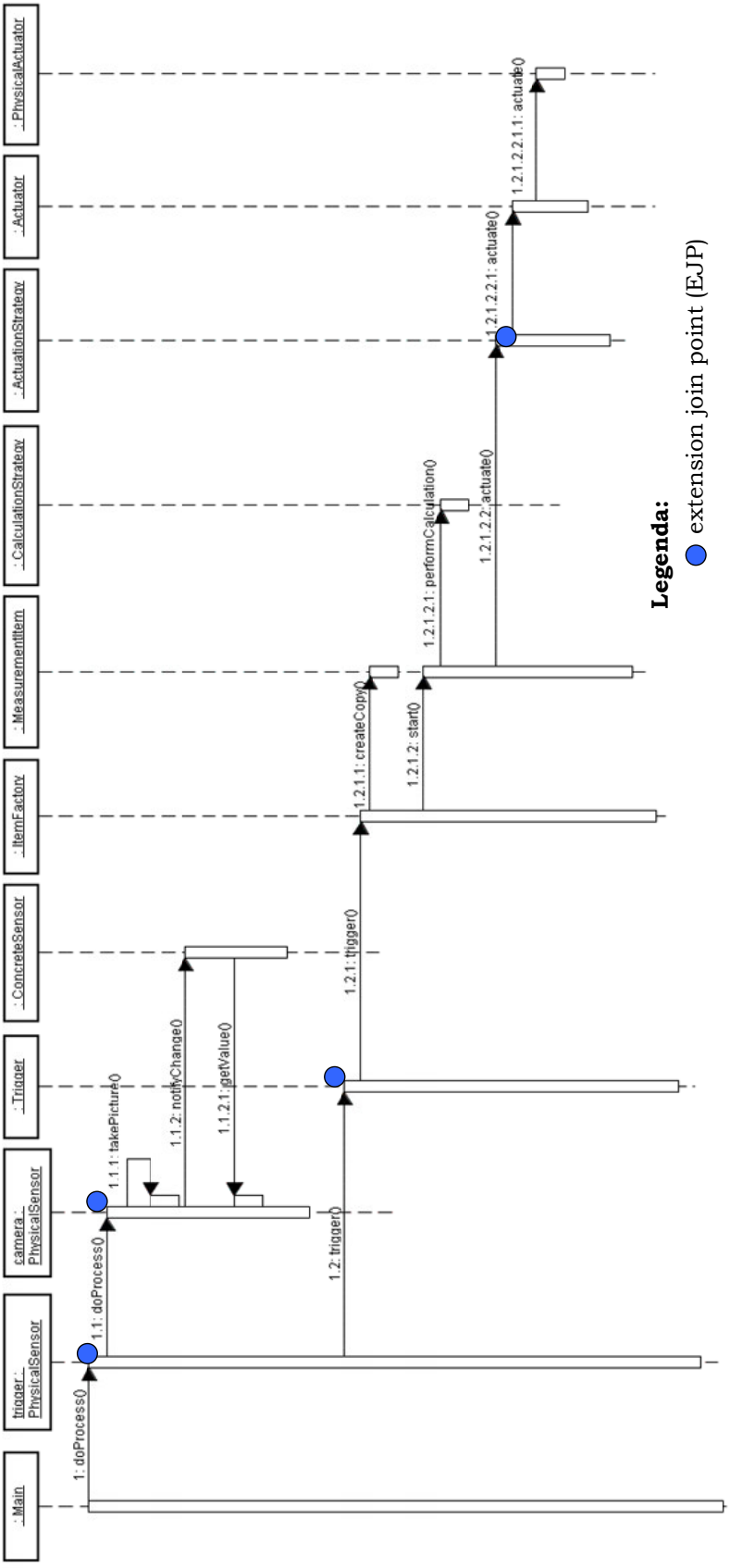

Figura 27. Diagrama de Seqüência do Framework Measurement

A Figura 28 mostra um exemplo de instância do framework para um sistema de análise de garrafas de cerveja (AGC), anteriormente apresentado em [17]. As seguintes classes foram criadas com tal propósito: (i) `Main` - classe principal da aplicação responsável por configurar e inicializar o framework; (ii) `BeerCanCamera` e `BeerCanTrigger` - subclasses de `PhysicalSensor` que representam, respectivamente, uma câmera e um trigger físico que atuam no processo de análise de cervejas; (iii) `BeerCanActuationStrategy` - define uma estratégia de atuação para o processo de análise que determina a ativação de atuadores em situações específicas; (iv) `BeerCanActuator` - representa um atuador físico concreto do sistema de AGC; (v) `Picture` - representa uma foto tirada pelo sensor; e (vi) `BeerCanCalculationStrategy` - determina o algoritmo para processamento de cada item medido pelos sensores.

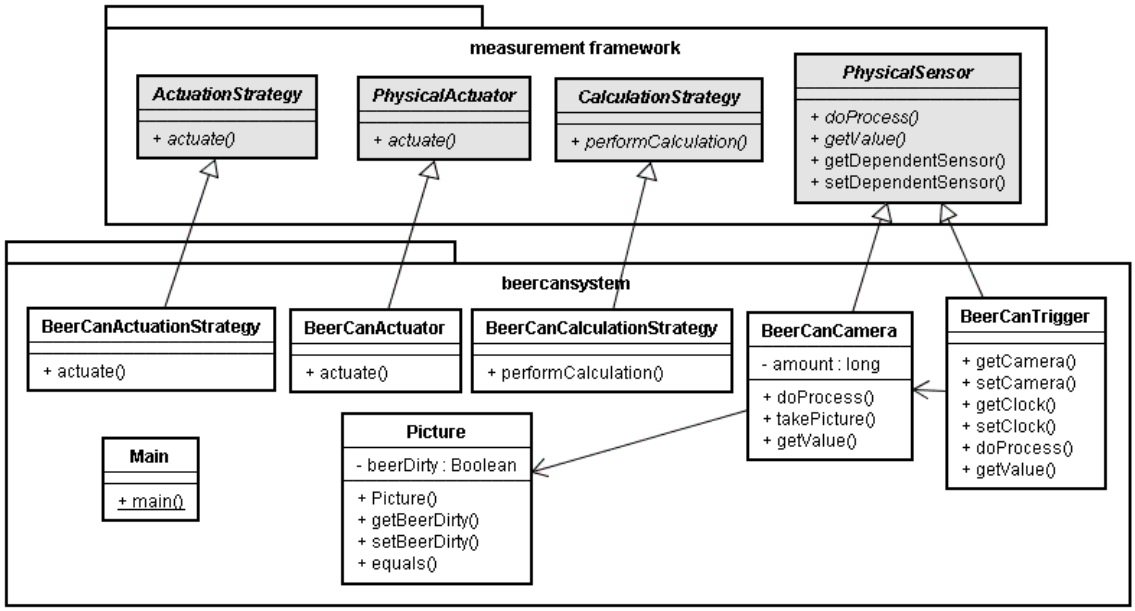

Figura 28. Exemplo de Instância do Framework *Measurement*

6.1.2. Pontos de Junção de Extensão

Durante a realização do estudo de composição do framework *Measurement* com outros frameworks [ref], foram identificadas que diversas composições tinham uma natureza transversal, requerendo dessa forma a interceptação de eventos internos específicos acontecendo na execução do framework

Measurement. Assim, os seguintes pontos de junção de extensão (EJPs) foram definidos para o framework *Measurement*;: (i) ativação de *triggers* – que representa o evento de inicialização do processo de análise da qualidade do produto; (ii) ativação de sensores – determina o evento de coleta de dados do produto; e (iii) ativação de acionadores (*actuators*) – esse evento representa o passo final de análise da qualidade do produto. Para expor esses pontos de junção de extensão, o aspecto EJP `MeasurementEvents` foi implementado em AspectJ, contendo três diferentes pontos de corte, que interceptam a execução de classes do framework. A Figura 29 mostra o código do aspecto EJP `MeasurementEvents`. A Figura 27 destaca no diagrama de seqüência do framework *Measurement*, os pontos de junção de extensão (EJPs) expostos.

```
01 public aspect MeasurementEvents {
02    // Event of data gathering by sensors
03    public pointcut physicalSensor(
04                       PhysicalSensor physicalSensor):
05       execution(public void PhysicalSensor+.doProcess())
06       && target(physicalSensor);
07
08    // Init the processing of an item
09    public pointcut triggerSW(Trigger triggerSW):
10          execution(public void Trigger.trigger()) &&
11          target(triggerSW);
12
13    // Finalize the processing of an item
14    public pointcut actuation(ActuationStrategy actuator):
15      execution(public void ActuationStrategy+.actuate(..))
16      && target (actuator);
17 }
```

Figura 29. Aspecto EJP `MeasurementEvents`

6.1.3. Aspectos de Integração

Nosso estudo de composição envolveu a composição do framework *Measurement* com outros três diferentes frameworks, sendo eles: (i) framework de interface gráfica (GUI) baseado na tecnologia Java Swing [44]; (ii) framework de análise estatística; e (iii) framework de persistência baseado no Hibernate[17]. Nosso objetivo foi avaliar diferentes tipos de composições entre frameworks de acordo

[17] Hibernate - Object/Relational Persistence and Query Service. URL: http://www.hibernate.org/. 2007..

com uma categorização de problemas de composição proposta por Mattsson et al [90, 91]. Cada integração entre o *Measurement* e os demais frameworks endereça um dos problemas de composição apresentados por aqueles autores. Nas subseções seguintes são apresentados os frameworks e os respectivos aspectos de integração que foram usados para prover uma composição com o framework *Measurement*.

6.1.3.1. Composição com o Framework GUI

O framework de interface gráfica (GUI), baseado em componentes Java Swing, permite a apresentação de dados da aplicação em diferentes tabelas visuais. A Figura 30 apresenta o diagrama de classe do framework GUI. Duas classes abstratas (`DataTableModel` e `ObjectTableModel`) definem abstrações de dados a serem apresentados em uma tabela Swing baseado, respectivamente, em um array ou uma lista de objetos. Essas classes herdam da classe `AbstractTableModel` da biblioteca Swing. Os dados da aplicação são apresentados visualmente usando a classe Swing `JTable`. Durante a instanciação do framework, o usuário deve também estender a classe `GUIFramework` e definir uma implementação para o seu método abstrato `createTables()`. A implementação desse método deve criar e retornar uma lista de objetos `JTable` configurado com seus respectivos objetos `AbstractTableModel`. Toda informação escrita em objetos `AbstractTableModel` são automaticamente apresentadas nos respectivos objetos `JTable` baseado em um protocolo de notificação implementado na biblioteca Swing [44].

Aspectos de Integração foram codificados para mostrar visualmente no framework GUI, detalhes dos itens que estão sendo processados pelo framework *Measurement*. O objetivo desses aspectos é interceptar a execução de métodos ocorrendo no framework *Measurement* e capturar informação para ser apresentada no framework GUI. Os EJPs do framework *Measurement* podem prover suporte para a implementação de tais aspectos de integração. A informação capturada pelos aspectos de integração é repassada para objetos `AbstractTableModel` do framework GUI. O protocolo de notificação entre objetos `AbstractTableModel` e `JTable` já implementado no framework GUI garante a visualização dos dados.

Para obter a referência para os objetos `AbstractTableModel` disponíveis, nossos aspectos de integração requereram a exposição de tais objetos. Dessa forma, a codificação de tais aspectos demandou a criação de um aspecto EJP do framework GUI, denominado `GUIEvents`, cujo objetivo é expor o método de criação `createTables()` do framework GUI. Esse método retorna um objeto do tipo `Map` que armazena os objetos `AbstractTableModel` criados. O aspecto EJP `GUIEvents` também expõe eventos relacionados a atualização de `ObjectTableModel` e `DataTableModel` no framework GUI. O código do aspecto EJP `GUIEvents` é apresentado na Figura 31.

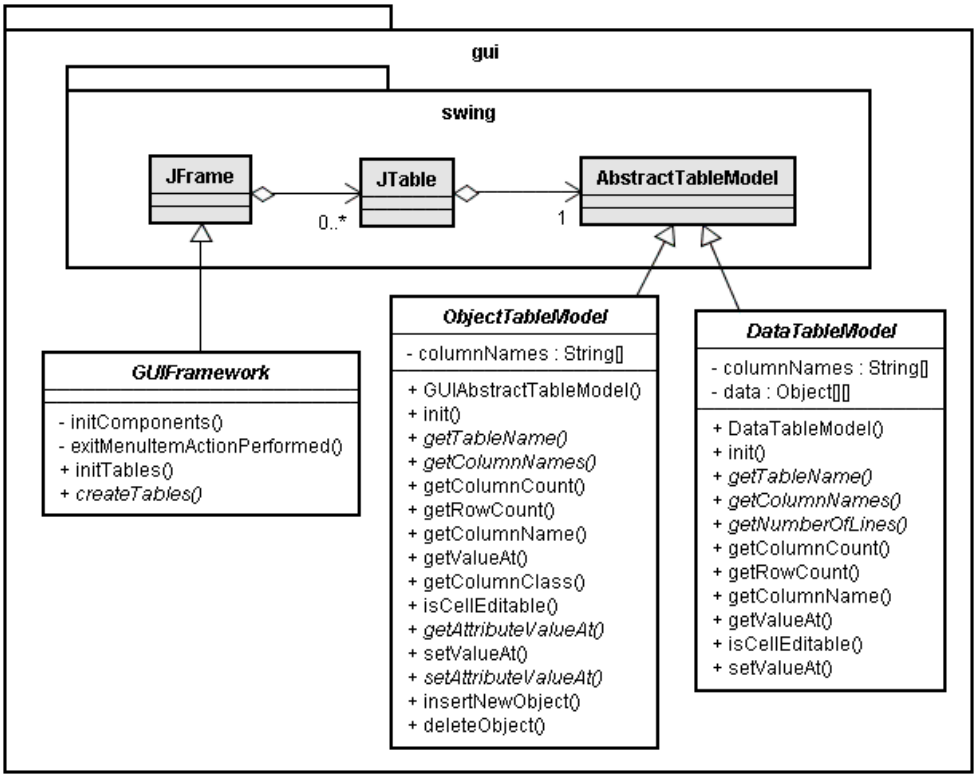

Figura 30. Framework GUI

A Figura 32 mostra os aspectos de integração. O aspecto `MeasurementGUIAspect` define o código comum que é sempre reusado quando é necessária uma composição entre os frameworks. Ele é responsável por interceptar os pontos de corte `triggerSW()` e `actuation()` do aspecto EJP `MeasurementEvents` de forma a capturar a informação dos itens de produtos que estão sendo processados por uma instância do framework *Measurement*. O subaspecto `BeerCanMeasurementGUIAspect` define o código variável que depende das instâncias criadas dos frameworks, tal como: (i) a inicialização da

instância do framework GUI durante a inicialização do framework *Measurement*; e (ii) a atualização de objetos `AbstractTableModel` específicos que irão apresentar os dados de uma instância do framework *Measurement*. A classe `BeerCanTableModel` é um exemplo de um objeto `TableModel` da aplicação de análise de qualidade de cervejas. Essa classe apresenta atributos de qualidade de cervejas que estão sendo processadas pelo framework *Measurement*.

```
01 public aspect GUIEvents {
02    public pointcut initializeTables():
03      execution(public Map GUIFramework+.createTables());
04
05    public pointcut updateObjectTable(Object object):
06      execution(public void
07          GUIAbstractTableModel.insertNewObject(Object))
08          && args(object);
09
10    public pointcut updateDataTable(Object object,
11                                    int row, int col):
12      execution(public void
13          DataTableModel.setValueAt(Object, int, int))
14          && args(object, row, col);
15 }
```

Figura 31. Aspecto EJP `GUIEvents`

Figuras 33 e 34 mostram o código parcial AspectJ dos aspectos de integração. O aspecto `MeasurementGUIAspect` define as seguintes funcionalidades: (i) um conjunto de métodos e pontos de corte abstratos que garantem a inicialização do framework GUI quando o framework *Measurement* é criado (linhas 5 e 9); (ii) ele salva uma referência para todos os objetos `AbstractTableModel` criados no framework GUI de forma a notificá-los quando ocorre atualização nos dados de itens de produto(linhas 2 e 13); e finalmente, (iii) ele intercepta a execução de métodos no framework *Measurement* para capturar informação a ser escrita nos objetos `AbstractTableModel`, tal como, o inicio do processamento de um item (linhas 17-29), a ativação de sensores e a finalização do processamento de um item nos objetos `ActuationStrategy` (linhas 35-44). O aspecto `MeasurementGUIAspect` também mantém internamente dados sobre os itens com o tempo inicial e final de processamento (linha 03). Esses dados são repassados para uma instância da classe `ProcessedItemTableModel` do framework GUI.

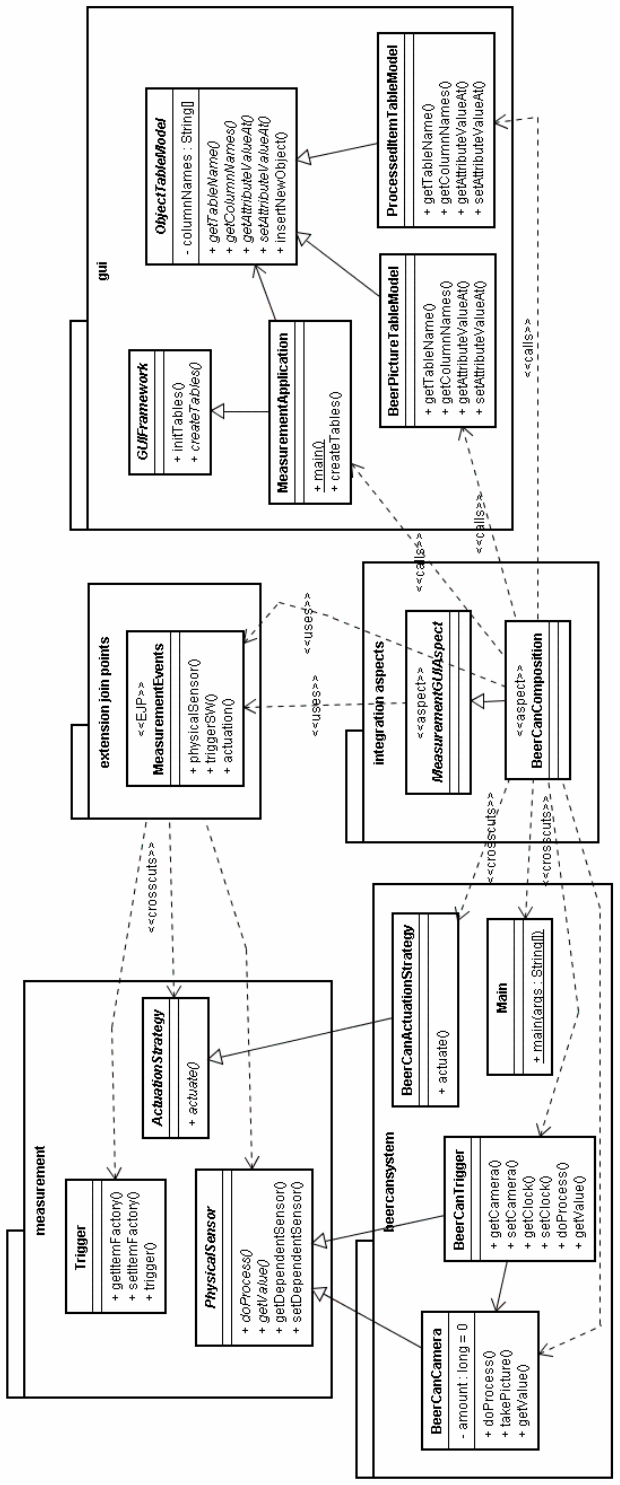

Figura 32. Integração entre os frameworks Measurement e GUI

```
01 public abstract aspect MeasurementGUIAspect {
02   private Map tables = null;
03   private Map itemsProcessingTime = new HashMap();
04
05   public abstract pointcut initIntegration();
06   before(): initIntegration() {
07       initIntegration();
08   }
09   public abstract void initIntegration();
10   ...
11   after() returning(Map tables):
12                GUIEvents.initializeTables(){
13      this.tables = tables;
14      this.initStandardTableModels();
15      this.initSpecificTableModels();
16   }
17   before(Trigger triggerSW):
18                MeasurementEvents.triggerSW(triggerSW){
19     this.createProcessedItem();
20     long threadId = Thread.currentThread().hashCode();
21     ProcessedItem currentItem =
22            (ProcessedItem) this.itemsProcessingTime.get(
23                                          new Long(threadId));
24     AbstractTableModel tableModel =
25       (AbstractTableModel) this.getTableModel(triggerSW);
26     if (currentItem != null && tableModelGeneral != null){
27           tableModelGeneral.insertNewObject(currentItem);
28     }
29   }
30   private void storeItemProcessingTime(){ ... }
31
32   public Map getItemsProcessingTime(){
33     return this.itemsProcessingTime;
34   }
35   after(ActuationStrategy actuator):
36            MeasurementEvents.actuation(actuator){
37     ...
38     ProcessedItem currentItem =
39            (ProcessedItem) this.itemsProcessingTime.get(
40                                           new Long(threadId));
41          currentItem.setEndTime(new Date());
42          // Updates the respective table model
43      ...
44   }
45 }
```

Figura 33. Aspecto de Integração `MeasurementGUIAspect`

O subaspecto `BeerCanMeasurementGUIAspect` especifica o ponto de corte que representa a inicialização do framework *Measurement* e implementa o método `initIntegration()` chamando o método `main()` que inicializa o framework GUI (linhas 3-8). Ele também implementa o método `getSpecificTableModel()` o qual retorna o objeto `AbstractTableModel` associado com um tipo específico de objeto (linhas 9-12). Esse método é chamado pelo método `getTableModel()` no aspecto `MeasurementGUIComposition`. O subaspecto `BeerCanMeasurementGUIAspect` pode também definir adendos e pontos de corte, que se responsabilizam pela atualização de objetos `AbstractTableModel` de uma instância do framework. A Figura 34 mostra, por exemplo, a definição de um adendo associado com o ponto de corte `physicalSensor()` o qual é exposto pelo aspecto EJP `MeasurementEvents`. Esse adendo atualiza o objeto `BeerCanTableModel` baseado em informação capturada pela classe `BeerCanCamera`, uma subclasse de `PhysicalSensor` (linhas 13-21).


```
01 public aspect BeerCanMeasurementGUIAspect extends
02                                  MeasurementGUIAspect {
03   public pointcut initIntegration():
04       execution(public static void BeerCanMain.main(..));
05
06   public void initIntegatrion(){
07      GUIApplication.main(null);
08   }
09   public AbstractTableModel getSpecificTableModel(
10                                Object object, Map tables){
11     ...
12   }
13   after (PhysicalSensor physicalSensor):
14         MeasurementEvents.physicalSensor(physicalSensor){
15     AbstractTableModel tableModel = (AbstractTableModel)
16                 this.getTableModel(physicalSensor);
17     Object object = physicalSensor.getValue();
18     if (object != null && tableModel != null) {
19        tableModel.insertNewObject(object);
20     }
21   }
22 }
```

Figura 34. Aspecto `BeerCanMeasurementGUIAspect`

6.1.3.2. Composição com o Framework de Estatística

O estudo de composição também contemplou a implementação de um framework de análise estatística simplificado. Esse framework oferece: (i) a classe facade `StatisticalFrameworkFacade` através da qual os serviços estatísticos são oferecidos; e (ii) duas classes abstratas, `StatisticalAlgorithm` e `Statistic`, que representam, respectivamente, pontos de extensão para implementar algoritmos estatísticos e a estatística calculada a partir do processamento do algoritmo.

O framework de Estatística foi usado para complementar a análise dos itens de produto processados pelo framework *Measurement*, de forma a calcular informações estatísticas relacionada a tal processo. As seguintes funcionalidades foram implementadas: (i) calcular o melhor e pior tempo de processamento de itens de produto; (ii) calcular o tempo médio de processamento de todos os itens de produto; e (iii) calcular o índice percentual de itens de produto de qualidade aceitável e não aceitável. A Figura 35 mostra as classes do framework de Estatística, assim como as classes necessárias para a sua instanciação no contexto do sistema de avaliação da qualidade de cervejas. Foram criadas classes para implementar as seguintes funcionalidades: (i) dois algoritmos de análise estatística (classes `ProcessingTimeAlgorithm` e `BeerCanQualityAlgorithm`) e (ii) os dados estatísticos concretos resultantes do processamento de tais algoritmos (classes `ProcessingTimeStatistic` e `BeerCanQualityStatistic`).

Para integrar a funcionalidade oferecida pelo framework de Estatística juntamente com os frameworks *Measurement* e GUI, foi criado um novo aspecto de integração, denominado `MeasurementGUIStatisticAspect`. A Figura 36 apresenta a relação de tal aspecto com os aspectos EJPs dos frameworks *Measurement* e GUI, e com o aspecto de integração `BeerCanMeasurementGUIAspect`. O aspecto `MeasurementGUIStatisticAspect` é responsável por configurar e usufruir dos serviços oferecidos pelo framework de Estatística para calcular informações de interesse da aplicação e, em seguida, apresentá-las visualmente no framework GUI, através da classe `ProcessingTimeTableModel`.

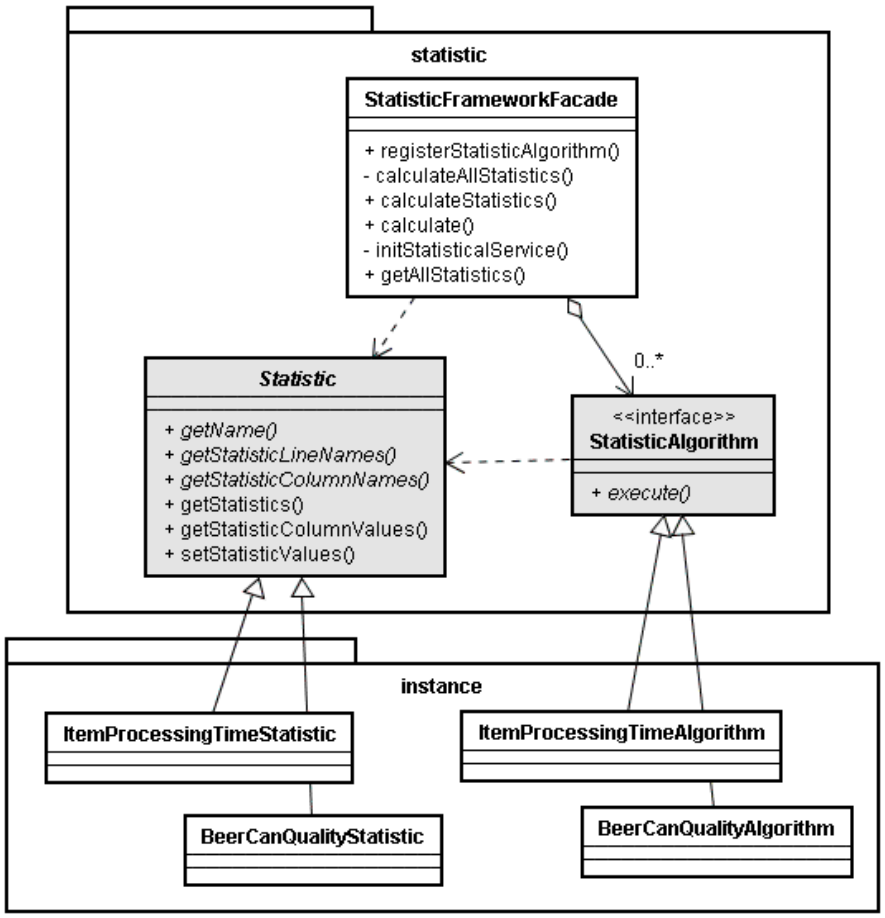

Figura 35. Framework de Estatística

A Figura 37 mostra o código do aspecto `MeasurementGUIStatisticAspect` responsável pela implementação das funcionalidade estatísticas. Ele possui as seguintes responsabilidades: (i) configurar o framework de Estatística para registrar algoritmos específicos a serem executados (linhas 13-22); (ii) criar os objetos `JTable` e respectivos objetos `AbstractTableModel` que representarão visualmente os dados estatísticos (linhas 23-28); e (iii) inicializar uma *thread* que calcula periodicamente os novos dados estatísticos através da invocação dos serviços do framework de Estatística, e atualiza os objetos `AbstractTableModel` com essas novas informações (linhas 29-52). Vale ressaltar que as informações dos itens sendo processados com seus respectivos instantes de processamento, são obtidas através de consulta ao aspecto de integração `BeerCanMeasurementGUIAspect` (linhas 50-51), definindo assim uma dependência entre os aspectos de integração. Essas informações são usadas para alimentar os algoritmos estatísticos.

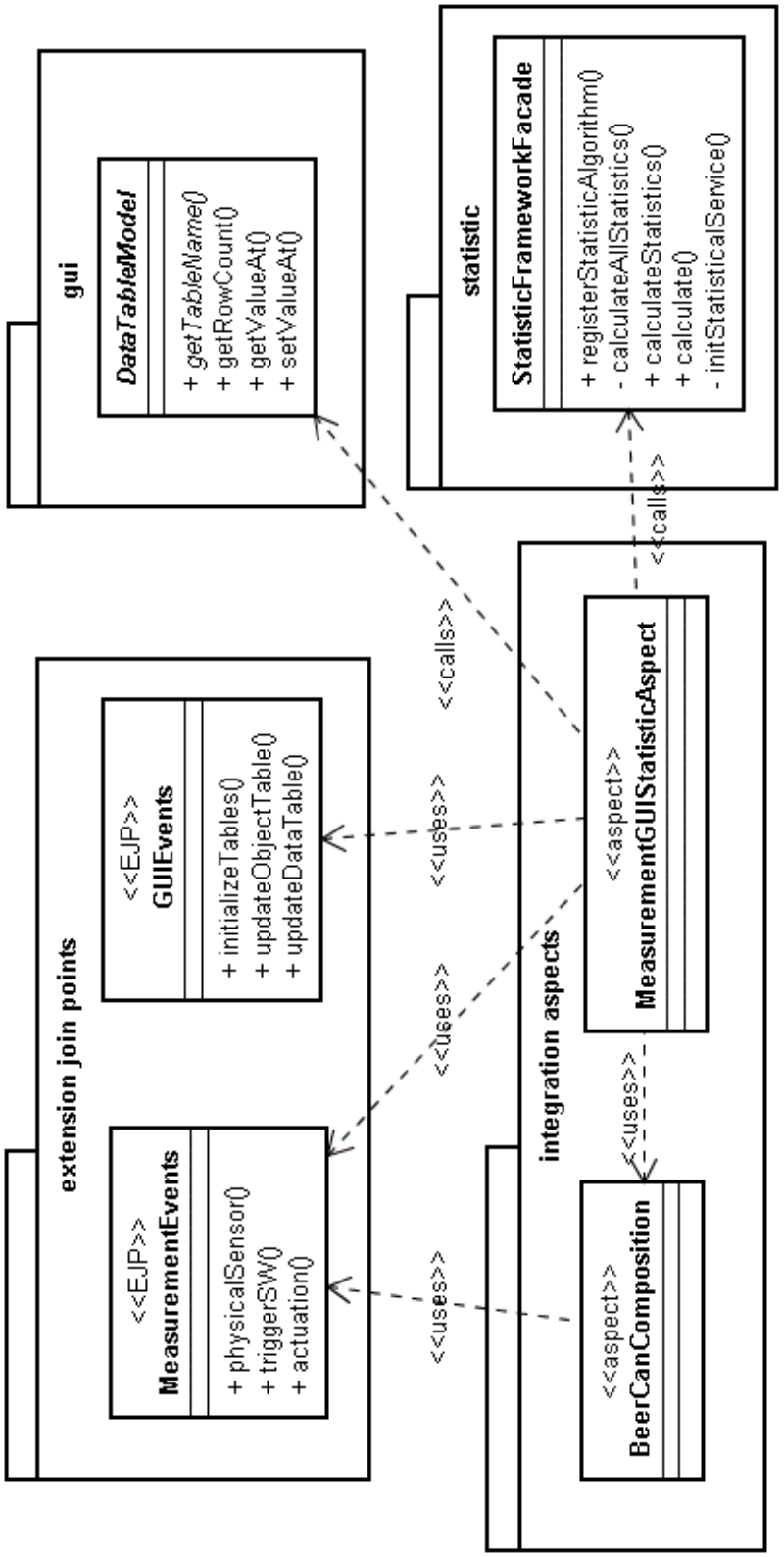

Figura 36. Integração entre os FWs Measurement, GUI e Estatística

```
01 public aspect MeasurementGUIStatisticAspect {
02   private Thread statisticalService = null;
03   private DataTableModel statisticTableModel = null;
04   public pointcut initIntegration():
05            MeasurementEvents.initApplication();
06   before(): initIntegration(){
07      this.configureStatisticFramework();
08   }
09   after() returning(Map tables):
10                GUIEvents.initializeTables(){
11     this.initStatisticTableModel(tables);
12   }
13   public void configureStatisticFramework(){
14      StatisticFrameworkFacade statisticFramework =
15                StatisticFrameworkFacade.getInstance();
16      StatisticAlgorithm algorithm =
17                new BeerProcessingTimeAlgorithm();
18      statisticFramework.registerStatisticAlgorithm(
19                  "Measurement", algorithm,
20                  this.getItemsProcessingTime());
21      this.initStatisticalService(10000);
22   }
23   public void initStatisticTableModel(Map tables){
24      this.statisticTableModel = new
25                          ProcessingTimeTableModel();
26      JTable table = new JTable(this.statisticTableModel);
27      tables.put("Processing Time Statistics", table);
28   }
29   private void initStatisticalService(final long delay){
30     if (statisticalService == null){
31        statisticalService = new Thread() { ...
32        };
33        statisticalService.start();
34      }
35   }
36   private void processStatistics(){
37      StatisticFrameworkFacade statisticFramework =
38                  StatisticFrameworkFacade.getInstance();
39      statisticFramework.updatePopulation("Measurement",
40                    this.getItemsProcessingTime());
41      Map statistics =
42      statisticFramework.getStatistiscs("Measurement");
43      this.updateStatistics(statistics);
44   }
45   private void updateStatistics(Map statistics){
46      // Update statistics in their respective table models
47      ...
48   }
49   private Collection getItemsProcessingTime(){
50      return BeerCanMeasurementGUIAspect.aspectOf().
51                    getItemsProcessingTime().values();
52   }
53   ...
54 }
```

Figura 37. Aspecto `MeasurementGUIStatisticAspect`

6.1.3.3. Composição com o Framework de Persistência

No nosso estudo de composição, o último interesse endereçado foi o de persistência. As seguintes informações foram persistidas: (i) dados dos itens de produto analisados pelo framework *Measurement*; e (ii) dados estatísticos oriundos desse processo de análise. Para endereçar a persistência de tais informações, foi utilizado o framework Hibernate. O Hibernate permite a definição de mapeamentos entre classes de negócio e tabelas de banco de dados, assim como oferece vários serviços para acesso e atualização do banco de dados.

Para possibilitar a persistência dos dados resultantes do processo de análise da qualidade de itens de produto, através do Hibernate, foi definida a classe `PersistenceFrameworkFacade`, responsável por prover os serviços de persistência e por agregar um conjunto de objetos de acesso a dados. Tais objetos de acessos a dados representam implementações concretas do padrão de projeto DAO (*Data Acess Object*) [1]. Cada DAO é responsável pela persistência de objetos de uma dada classe do domínio do sistema. No framework de persistência, a classe abstrata `EntityDAO` especifica um conjunto de métodos de persistência a serem implementados por DAOs concretos. Essa classe herda da classe `HibernateDAOSupport` do framework Spring [65], de forma a reusar serviços de banco de dados oferecidos pelo framework Hibernate, tais como, consultas simples de acesso ao BD, gerenciamento de conexões e demarcação de transações.

A Figura 38 mostra os aspectos de integração e as classes DAOs responsáveis pela implementação do interesse de persistência. Para capturar as informações a serem persistidas, o aspecto de persistência `PersistenceAspect` intercepta: (i) o ponto de execução do framework GUI, exposto pelo aspecto EJP `GUIEvents`, que representa a atualização de dados em objetos `ObjectTableModel`; e (ii) o ponto de execução do framework de Estatística, exposto pelo aspecto EJP `StatisticEvents`, o qual expõe a execução de algoritmos estatísticos. Essas informações coletadas pelo aspecto `PersistenceAspect` são repassadas para a classe `PersistenceFrameworkFacade`, que se encarrega de chamar a classe DAO apropriada para persistir no banco de dados tais informações. A configuração dos DAOs a serem usados pelo framework de persistência é feita também pelos

aspectos de integração, o aspecto `PersistenceAspect` configura DAOs que serão usados em diferentes configurações de composição, enquanto seu subaspecto `BeerCanPersistenceAspect` se responsabiliza pela configuração de DAOs específicos.

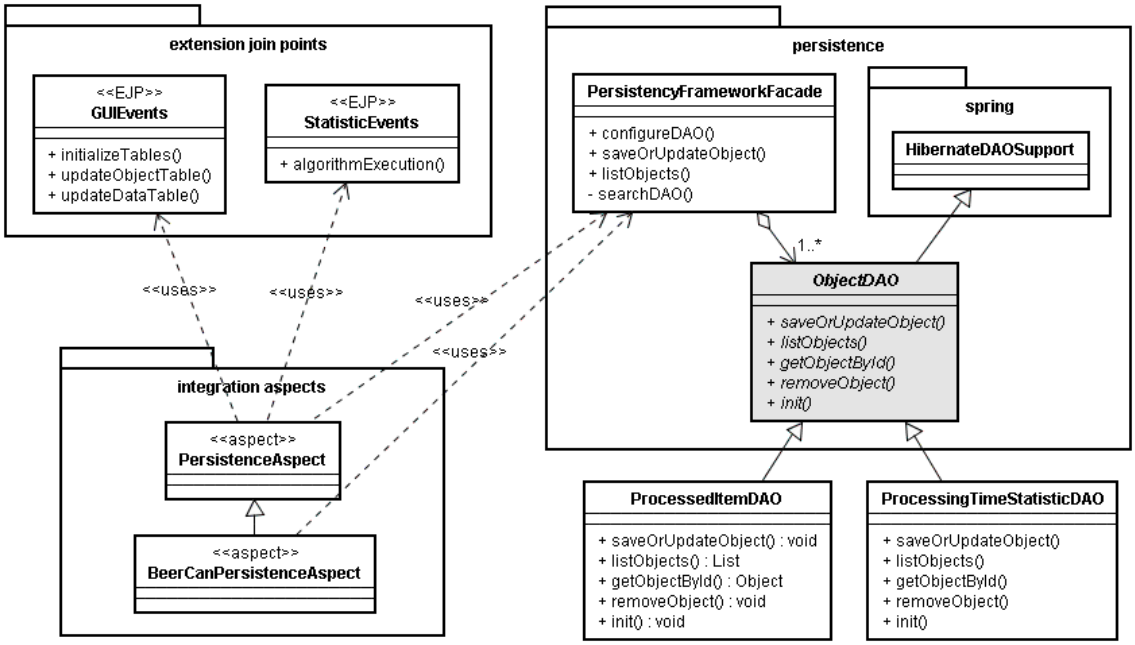

Figura 38. Integração entre os FWs GUI, de Estatística e de Persistência

As Figuras 39 e 40 mostram o código dos aspectos de integração de persistência. O aspecto `PersistenceAspect` define os seguintes pontos de corte: (i) `initializePersistenceService()` – o qual é usado para disparar a inicialização do framework de persistência com a configuração de seus respectivos DAOs (linha 4). Esse ponto de corte é concretizado por subaspectos; (ii) o ponto de corte `processedItemsTableModel()` – o qual intercepta a atualização de objetos `ObjectTableModel` no framework GUI, de forma a persistir tais informações (linhas 19-24); e (iii) `statisticalDataPersistence()` – responsável por interceptar a execução de algoritmos de estatística no framework respectivo, de forma a coletar os dados estatísticos resultantes de tal processamento e invocar o método `saveOrUpdateObject()` do framework de persistência (linhas 26-31). O subaspecto `BeerCanPersistenceAspect`, por sua vez, concretiza o ponto de corte de inicialização do framework GUI (linhas 3-5) e inicializa, caso existam, novos DAOs concretos (linhas 7-11).

```
01 public abstract aspect PersistenceAspect {
02   PersistencyFrameworkFacade persistenceFramework = null;
03
04   public abstract pointcut initializePersistenceService();
05
06   after(): initializePersistenceService(){
07     persistenceFramework =
08               PersistencyFrameworkFacade.getInstance();
09     this.initCommonDAOs();
10     this.initSpecificDAOs();
11   }
12   public void initCommonDAOs(){
13     this.persistenceFramework.configureDAO(
14               ItemProcessingTime.class.getName(),
15               new ProcessedItemDAO());
16   }
17   public abstract void initSpecificDAOs();
18
19   public pointcut processedItemsTableModel(Object object):
20                GUIEJPs.updateObjectTable(object);
21
22   after(Object object): processedItemsTableModel(object){
23     this.persistenceFramework.saveOrUpdateObject(object);
24   }
25
26   public pointcut statisticalDataPersistence():
27      StatisticEvents.algorithmExecution();
28
29   after() returning(Statistic statistic):
30               statisticalDataPersistence(){
31    this.persistenceFramework.saveOrUpdateObject(statistic);
32
33   }
34 }
```

Figura 39. Aspecto de Integração `PersistenceAspect`

```
01 public aspect BeerCanPersistenceAspect
02                          extends PersistenceAspect {
03   public pointcut initializePersistenceService():
04      execution(public static void
05                       MeasurementApplication.main(..));
06
07   public void initSpecificDAOs(){
08      this.persistenceFramework.configureDAO(
09             BeerProcessingTimeStatistic.class.getName(),
10             new ProcessingTimeStatisticDAO());
11   }
12 }
```

Figura 40. Aspecto de Integração `BeerCanPersistenceAspect`

6.1.4. Modelo Generativo OA

O modelo generativo do framework *Measurement* integrado com os frameworks de GUI, Estatística e Persistência é apresentado na Figura 41. O modelo de arquitetura agrega os quatro componentes principais que representam cada um dos frameworks da composição, são eles: measurement, gui, statistic, e persistence. Cada um deles agrega: (i) as classes que implementam o núcleo do framework; (ii) templates úteis para a instanciação do framework que são especializações de classes abstratas ou interfaces que representam os pontos flexíveis do framework; e (iii) seus respectivos EJPs, os quais expõem eventos/estados específicos oriundos de colaborações de classes do framework, e que podem ser usados para prover alguma integração transversal com um outro framework/componente. O componente measurement, por exemplo, define os templates `ActuatorTemplate`, `SensorTemplate`, `ActuationStrategyTemplate` e `CalculationStrategyTemplate` para a criação de efetuadores, sensores e estratégias relacionadas ao processo de análise dos produtos no framework *Measurement*. O aspecto EJP `MeasurementEvents` também é definido dentro de tal componente. Por questões de espaço apenas parte dos elementos de implementação de cada componente são apresentados.

Para os demais componentes (gui, statistic, persistence), foram também definidos no modelo de arquitetura: (i) um sub-componente integration - responsável por agregar aspectos e classes relacionados com a integração entre os frameworks; e (ii) classes diretamente relacionadas com a instanciação do framework para o contexto da composição com o framework *Measurement*. Os aspectos e classes de integração entre os frameworks *Measurement* e GUI, por exemplo, são definidos dentro do sub-componente integration do componente gui. As classes `SensorTableModel` e `ActuatorTableModel` representam uma instanciação do framework GUI para representar informações sobre o processo de análise de qualidade dos produtos.

O modelo de característica da composição dos frameworks expõe as variabilidades de cada um deles para serem escolhidas pelos engenheiros de aplicação. No que se refere ao processo de medição, o usuário pode definir sensores (Sensor) e acionadores (Actuator) concretos a serem usado em tal

processo. A característica GUI permite ao usuário definir que informações ele deseja visualizar do processo de medição, tais como, informações de itens já processados por sensores e efetuadores (Sensor e Actuator Visualization) e/ou informações detalhadas do processo de análise de qualidade do produto (Product Information Visualization). A característica Statistic permite ao usuário decidir pela visualização ou não de informações estatísticas sobre os itens sendo processados pelo framework *Measurement*. Finalmente, a característica Persistence permite escolher que informação manipulada pelos frameworks se deseja persistir em banco de dados.

No modelo de configuração da composição dos frameworks de *Measurement*, GUI, Estatística e Persistência, diferentes relações de dependência são criadas entre os elementos de implementação dos frameworks e características. Templates de classes representando sensores, efetuadores e estratégias relacionadas ao processo de medição, dependem das respectivas variabilidades existentes no modelo de característica. O template `ActuatorTemplate`, por exemplo, depende explicitamente da criação de uma característica Actuator. Cada um dos componentes gui, statistic e persistence dependem, respectivamente, da seleção das características GUI, Statistic e Persistence. Isso significa que todos os elementos de implementação internos a tais componentes e que não possuem alguma outra relação de dependência explícita, serão incluídos em um produto instanciado sempre que as características dos quais dependem forem selecionadas. Finalmente, os diferentes elementos que implementam as alternativas de interface gráfica (`SensorTableModel`, `ActuatorTableModel`, `ItemTableModel`) e persistência (`ProcessedItemDAO`, `ProcessingTimeStatisticDAO`) também dependem explicitamente das características alternativas de GUI e Persistence.

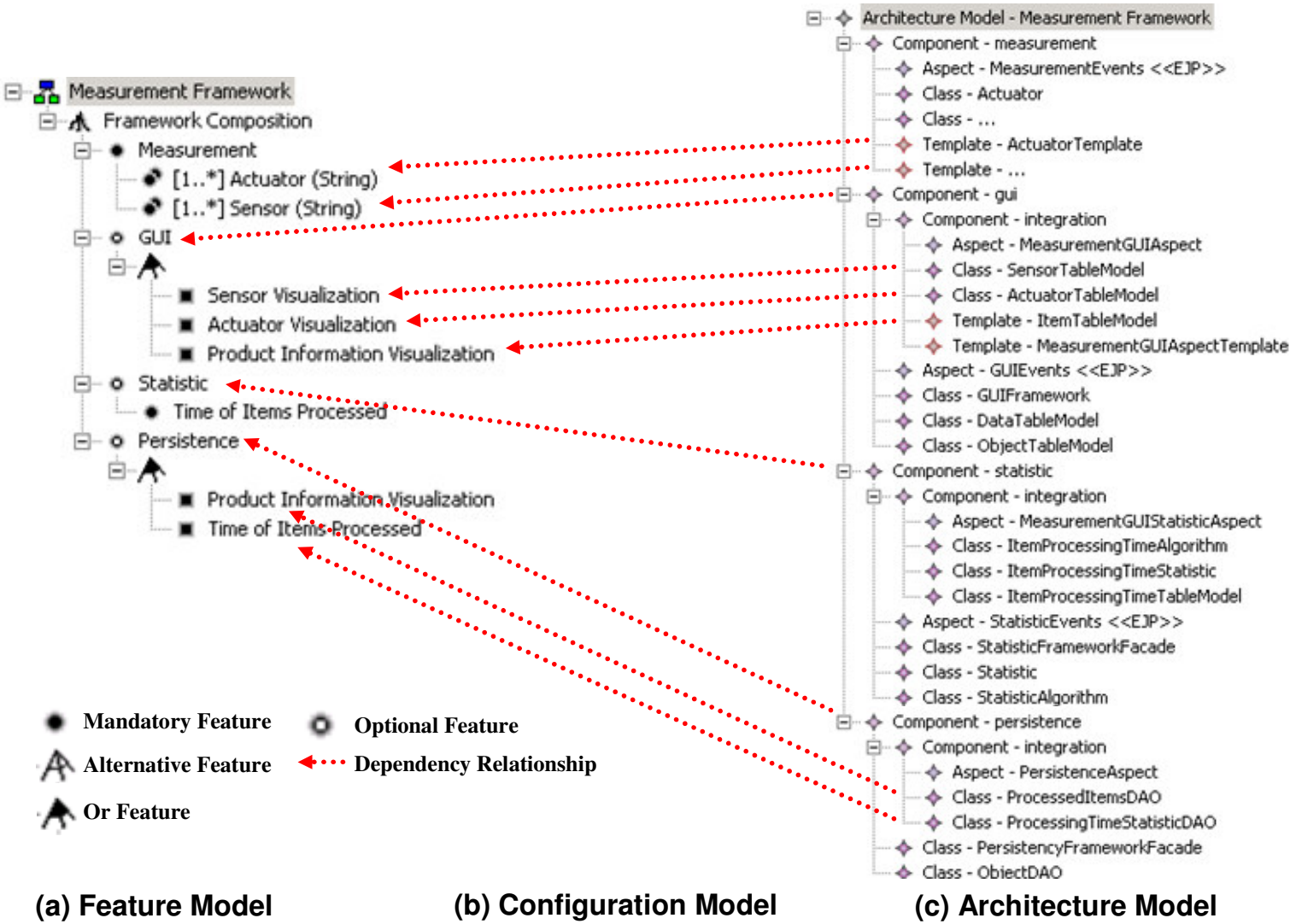

Figura 41. Modelo Generativo da Composição dos Frameworks

6.2. AspectT

Nossa abordagem foi também aplicada na reestruturação do framework orientado a aspectos AspectT [46, 50, 80]. AspectT é um framework usado na implementação de arquiteturas de agentes que endereçam diferentes propriedades. Ele foi desenvolvido para endereçar uma melhor modularização do projeto e implementação de diferentes tipos de agentes que precisam endereçar um conjunto de funcionalidades e propriedades (interação, adaptação, autonomia, aprendizado, mobilidade) que são transversais entre si.

Assim, o framework AspectT busca a modularização de propriedades transversais encontrados na definição da arquitetura de agentes, oferecendo mais flexibilidade para a composição transversal entre tais propriedades. A separação de interesses transversais oferecida pelo AspectT pode também trazer os benefícios: (i) de diminuição de linhas de código do sistema e (ii) de facilitar a manutenibilidade e reusabilidade [46, 50] do sistema.

6.2.1. Núcleo do Framework

O núcleo do AspectT define um conjunto de serviços básicos do agente para manipulação do seu conhecimento e para interação com o mundo externo. Cada agente contém crenças, objetivos e planos. As crenças do agente definem informação sobre o próprio agente e sobre o ambiente no qual ele está inserido. Para alcançar seus objetivos, um agente executa planos específicos. Durante a execução do plano, o agente manipula suas crenças. Cada agente também mantém um conjunto de sensores e efetuadores os quais habilitam e permitem sua interação com o mundo externo.

A Figura 42 apresenta um diagrama de classes parcial do núcleo do framework AspectT. A classe `Agent` define o comportamento básico de um agente e agrega instâncias das classes `Goal` e `Plan`. Essas últimas, por sua vez, especificam a estrutura e comportamento geral que deve existir para a definição de objetivos e planos do agente. A classe `Agent` também mantém atributos e métodos para a troca de mensagens com o ambiente, através da classe `Interaction`. Essa classe é responsável pela definição de: (i) atributos `inbox` e

`outbox` para armazenamento de mensagens recebidas e enviadas, respectivamente; e (ii) um conjunto de sensores (subclasses de `Sensor`) e efetuadores (subclasses de `Effector`) para interação com o ambiente externo. A Figura 43 mostra o diagrama de seqüência de recebimento e envio de mensagens do agente.

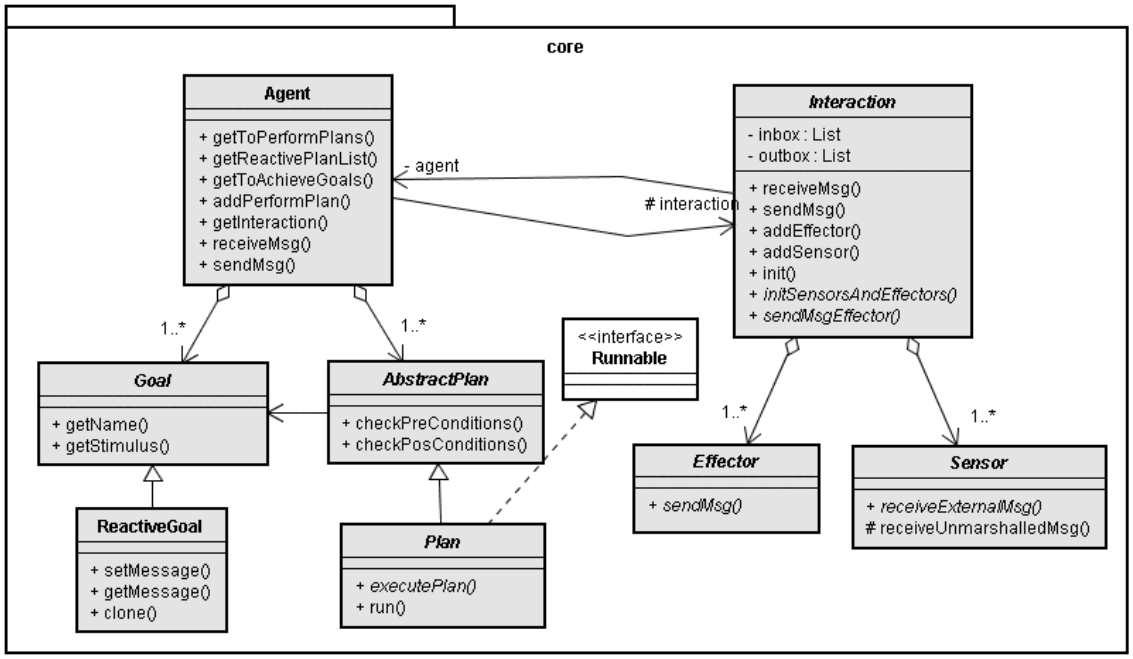

Figura 42. Núcleo do Framework AspectT

Várias classes do núcleo do AspectT representam pontos flexíveis do framework os quais podem ser estendidos para implementar agentes específicos. A classe `Agent` pode ser estendida para definir novos tipos de agentes. Crenças específicas de determinados tipos de agentes podem ser implementadas diretamente como classes de domínio que são agregadas pelas subclasses de `Agent`. As classes `Goal`, `ReactiveGoal`, `Plan` e `ReactivePlan` podem, por sua vez, ser especializadas para definir planos e objetivos específicos de um dado tipo ou papel de agente. Finalmente, as classes `Interaction`, `Sensor` e `Effector` definem uma estrutura geral para a criação dos elementos de comunicação do agente. Subclasses das mesmas devem ser criadas para definir sensores e efetuadores usados por um dado agente.

A Figura 44 mostra um exemplo de instância do framework AspectT para um sistema de gerência do processo de revisão de artigos de conferência,

denominado Expert Committtee (EC) [46]. O sistema EC define agentes do usuário que representam assistentes de software para auxiliar em atividades de gerência e distribuição de artigos. As seguintes classes foram definidas para instanciação do framework AspectT: (i) classe `ResearcherUserAgent` – implementa um agente do usuário que representa pesquisadores, tais agentes podem assumir os papéis *Chair* e *Reviewer* no contexto do EC; (ii) classes `BlackboardSensor` e `BlackboardEffector` – implementam sensores e efetuadores para comunicação via blackboard [19] entre agentes; (iii) subclasses das classes `ReactiveGoal` e `ReactivePlan`, são também criadas para endereçar objetivos e planos a serem alcançados pelos papéis *Chair* e *Reviewer*.

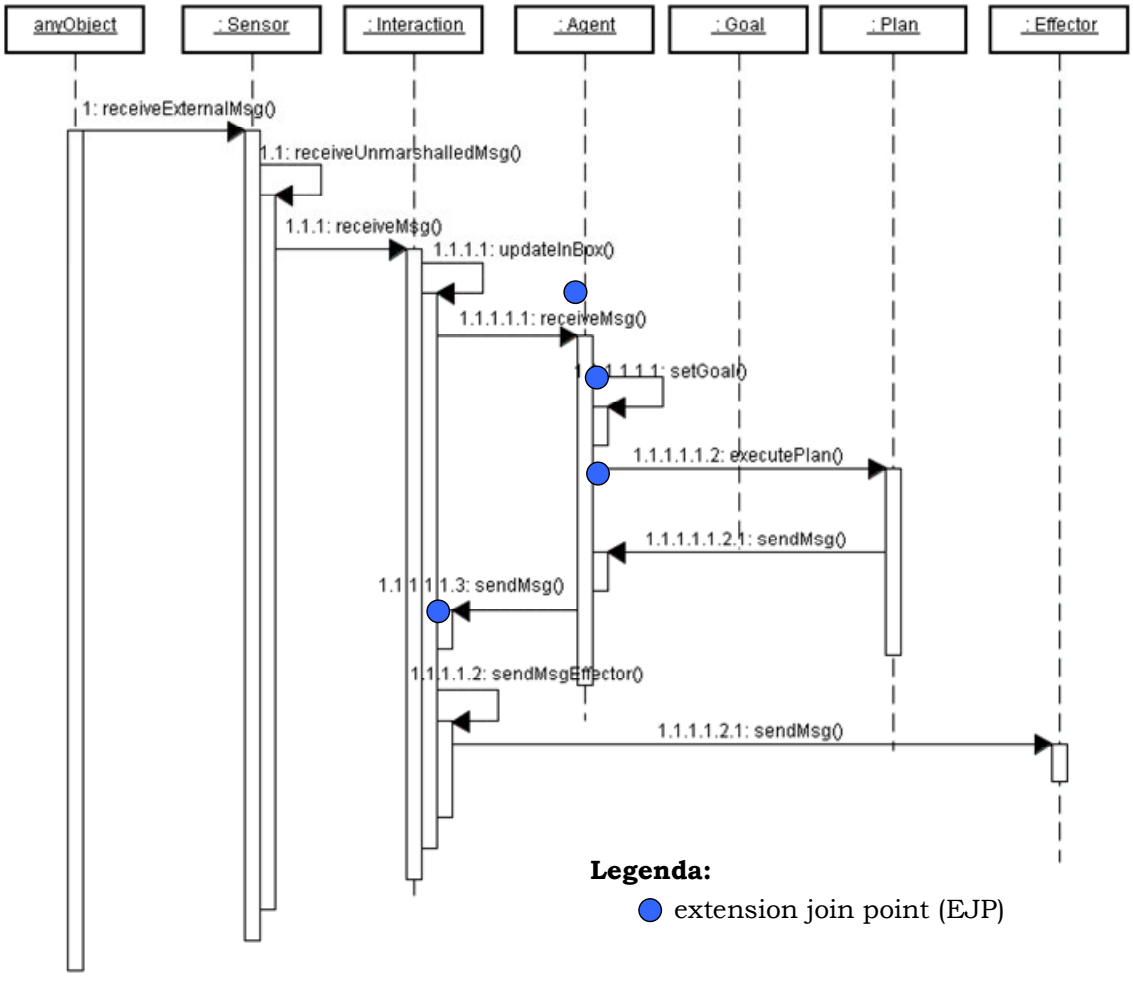

Figura 43. Diagrama de Seqüência do Framework AspectT

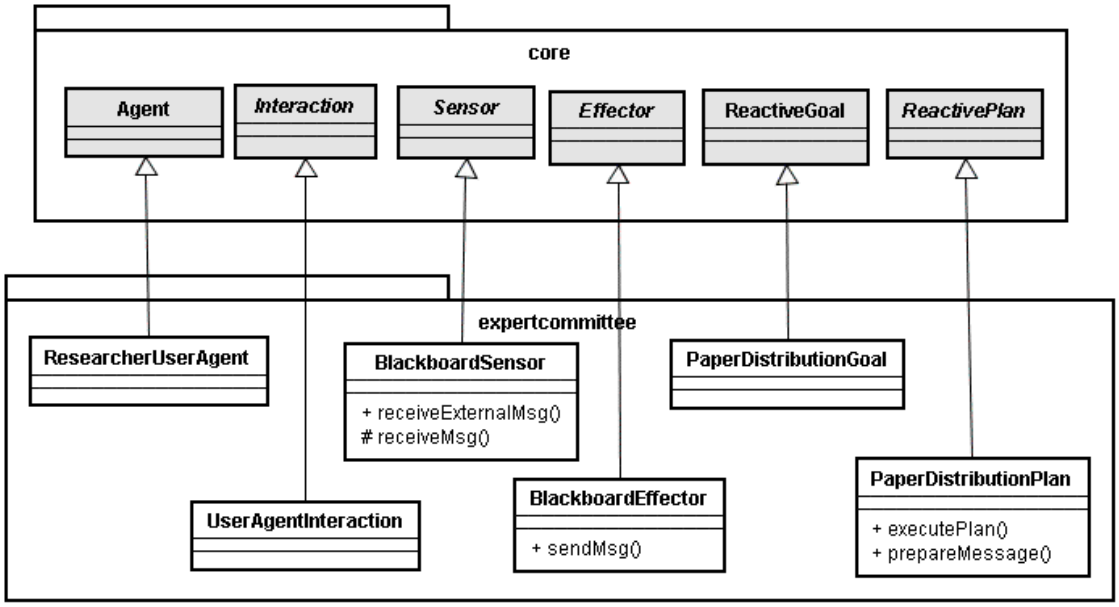

Figura 44. Exemplo de Instância do Framework AspectT

6.2.2. Pontos de Junção de Extensão

O framework AspectT expõe os seguintes EJPs: (i) eventos de recepção e envio de mensagens; (ii) evento de instanciação de objetivos do agente; e (iii) evento de execução de planos. A Figura 45 mostra o código do aspecto EJP `AspectTEvents` do framework AspectT, com os respectivos pontos de corte que expõem eventos de interesse para implementação de aspectos do núcleo e de extensão do framework.

```
01 public aspect AspectTEvents {
02   public pointcut messageReceiving(Agent agent, Message msg):
03     args(msg) && this(agent)
04     && execution(void Agent.receiveMsg(Message));
05
06   public pointcut goalCreation(Agent agent, Goal agentGoal):
07     args(agentGoal) && this(agent)
08     && execution(void setGoal(Goal));
09
10   public pointcut planExecution(Plan plan):
11     call(public void Plan+.executePlan(..))
12      && target(plan);
13
14   public pointcut messageSending(Agent agent, Message msg):
15       execution(void sendMsg(Message msg, Agent agent))
16       && args(msg, agent);
17 }
```

Figura 45. Aspecto EJP `AspectTEvents`

6.2.3. Aspectos do Núcleo

A funcionalidade básica oferecida pelo núcleo do framework AspectT contempla também a definição de dois aspectos do núcleo, os quais implementam duas características obrigatórias do agente, são elas: (i) Adaptação – essa característica define mudanças a serem realizadas nas crenças ou comportamento do agente, a partir da percepção de eventos ocorridos no ambiente ou na própria execução do agente; e (ii) Autonomia – determina a instanciação de objetivos a partir da ocorrência de eventos internos e externos ao agente. As subseções seguintes apresentam os aspectos do núcleo que foram implementados para endereçar as características de Adaptação e Autonomia.

6.2.3.1. Adaptação

A característica de Adaptação é implementada no AspectT pelo aspecto abstrato `Adaptation` e por um conjunto de subaspectos de tal aspecto que especificam comportamento de adaptação específico para um dado tipo ou papel de agente. Dois tipos de adaptação são endereçados por esses aspectos: (i) adaptação de crenças – responsável por interpretar mensagens recebidas do ambiente e manipular/atualizar as crenças do agente baseado em informações contidas em tais mensagens; e (ii) adaptação de planos – determina o plano que o agente deve executar quando determinados objetivos precisam ser alcançados. A Figura 46 apresenta a estrutura geral de solução do AspectT para endereçar a característica de Adaptação, através do aspecto `Adaptation` e seus subaspectos.

A Figura 47 apresenta o código do aspecto abstrato `Adaptation` e os elementos principais que ele interage. A adaptação de crenças desse aspecto é definida através da interceptação do evento de recepção de mensagens – ponto de corte `messageReceiving()` – definido no aspecto EJP `AspectTEvents` (linhas 2-3). Advices e métodos específicos são responsáveis pela atualização de crenças do agente a partir da recepção de mensagens do ambiente (linhas 5-18). A adaptação de planos do aspecto `Adaptation` intercepta o evento de instanciação de objetivos do agente – ponto de corte `goalCreation()` (linhas 20-21), assim

como de falha na execução de planos – ponto de corte `exceptionalPlanExecution`) (linhas 39-42). O objetivo é definir novos planos, através de chamadas aos métodos `findPlan()` e `findSpecificPlan()` (linhas 34-37), para serem executados pelos agentes durante essas situações específicas. O aspecto `Adaptation` pode ser especializado para permitir a definição de adaptação de crenças e planos específicos para um dado tipo ou papel de agente. Os métodos abstratos `adaptSpecificBelief()` e `findSpecificPlan()` podem ser redefinidos por subaspectos para implementar tais funcionalidades. A seção 6.2.4.3 apresenta exemplos de subaspectos da característica de Adaptação.

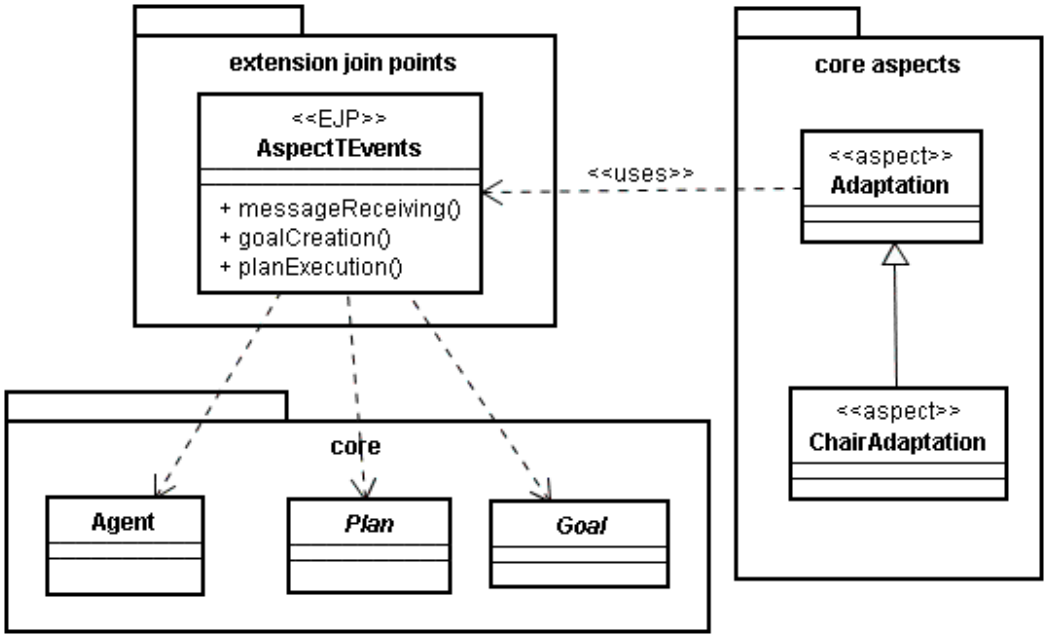

Figura 46. Aspecto do Núcleo Adaptation

```
01 public abstract aspect Adaptation {
02   pointcut beliefAdaptation(Agent agent, Message msg):
03          AspectTEvents.messageReceiving(agent, msg);
04
05   after(Agent agent, Message msg):
06          beliefAdaptation(agent, msg) {
07      adaptBelief(agent, msg);
08   }
09   public void adaptBelief(Agent agent, Message msg){
10     adaptGeneralBelief(agent, msg);
11     adaptSpecificBelief(agent, msg);
12   }
13   public void adaptGeneralBelief(Agent agent, Message msg){
14     // Update the agent belief, based on the received msg
15     ...
16   }
17   public abstract void adaptSpecificBelief
18                            (Agent agent, Message msg);
19
20   pointcut planAdaptation(Agent agent, Goal agentGoal):
21         AspectTEvents.goalCreation(agent, agentGoal);
22   after(Agent agent, Goal agentGoal) returning() :
23          planAdaptation(agent, agentGoal) {
24     Hashtable agentPlans = new Hashtable();
25     Plan plan = findSpecificPlan(agent, agentGoal);
26     if (plan == null) {
27       agentPlans = agent.getReactivePlanList();
28       plan = findPlan(agent, agentGoal, agentPlans);
29     }
30     if (plan != null) {
31       agent.addPerformPlan(plan);
32     }
33   }
34   public abstract Plan findSpecificPlan(Agent agent,
35                                         Goal agentGoal);
36   public Plan findPlan(Agent agent, Goal agentGoal,
37                        Hashtable agentPlans) { ... }
38
39   pointcut exceptionalPlanExecution(Plan plan):
40       AspectTEvents.planExecution(plan);
41   after(Plan plan) throwing(FailedPlanException exception):
42      exceptionalPlanExecution(plan) { ... }
43
44   protected pointcut planFinalization(Plan plan):
45      AspectTEvents.planExecution(plan);
46   after(Plan plan) returning():
47         planFinalization(plan) { ... }
48 }
```

Figura 47. Aspecto do Núcleo `Adaptation`

6.2.3.2. Autonomia

A característica de Autonomia endereça a funcionalidade de instanciação e gerência de objetivos. Ela lida com três tipos de objetivos: (i) reativos – aqueles que são instanciados a partir de requisições externas de agentes ou demanda do próprio ambiente; (ii) proativos – são criados a partir da ocorrência de eventos internos (tais como, finalização de execução de planos ou alcance de determinado estado de execução); e (iii) de decisão – são instanciados devido a eventos internos ou externos e usados para decidir se objetivos reativos ou proativos devem ser criados. A característica Autonomia também se responsabiliza pela definição de uma estratégia de concorrência para execução dos planos do agente.

A Figura 48 mostra os aspectos e classes responsáveis pela implementação da característica Autonomia. O aspecto `Autonomy` é o responsável principal pela extensão do núcleo do framework AspectT para endereçar a propriedade de Autonomia. Ele é usado para definir o comportamento básico de: (i) instanciação e gerência de objetivos; e (ii) execução concorrente de planos. Tipos de agentes mais sofisticados podem estender o aspecto `Autonomy` para definir uma gerência de objetivos específica para um dado tipo ou papel de agente.

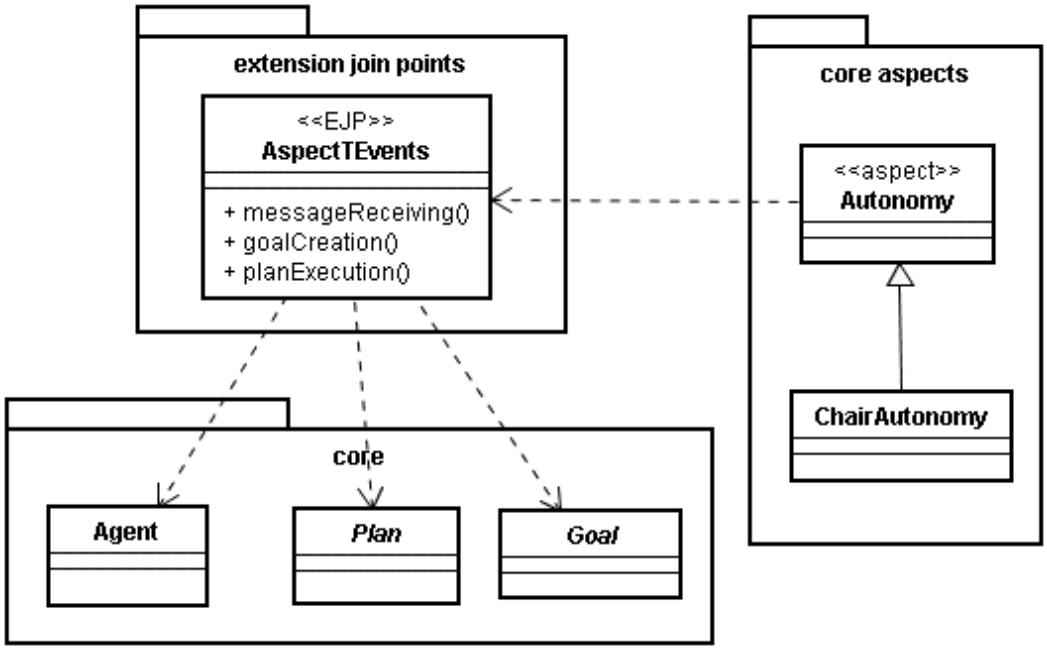

Figura 48. Aspecto do Núcleo Autonomy

O aspecto `Autonomy` demanda a instanciação de objetivos reativos a partir da interceptação do evento de recebimento de mensagens do agente, exposto pelo ponto de corte `messageReceiving()` do aspecto EJP `AspectTEvents` (linhas 33-54). Caso necessário, ele (ou seus sub-aspectos) pode também especificar comportamento para instanciação de objetivos proativos, a partir do monitoramento de eventos internos do agente. Nesse caso diferentes eventos internos expostos pelos aspectos EJPs podem ser utilizados. Adendos e métodos específicos (tais como, `instantiateSpecificReactiveGoal()` e `instantiateSpecificProactiveGoal()`) definem o comportamento propriamente dito de instanciação de objetivos na ocorrência de tais eventos. A autonomia de decisão do agente é definida pelo método `makeDecision()` do aspecto `Autonomy` (linhas 40-44). Esse método verifica se é necessário executar algum plano de decisão na ocorrência de eventos internos ou externos. Subaspectos de `Autonomy` podem também ser codificados para definir comportamento de instanciação de objetivos (reativos, proativos e de decisão) específicos de um dado tipo ou papel de agente, através da concretização de vários métodos abstratos, tais como: `instantiateSpecificReactiveGoal()`, `instantiateSpecificProactiveGoal()` e `makeSpecificDecision()`.

Finalmente, o aspecto `Autonomy` também define um *Active Object* [86] (linhas 20-31). Ele é usado para: (i) monitorar a lista de planos do agente que foram selecionados pela característica de adaptação para serem executados; e (ii) selecionar uma estratégia de concorrência que permite a execução de planos usando diferentes *threads*.

```
01 public abstract aspect Autonomy {
02   private GoalFinder goalFinder;
03   ...
04   /* Goal Creation */
05   protected abstract pointcut agentInstantiation
06                                   (Agent agent);
07   after (Agent agent): agentInstantiation(agent) {
08     goalFinder = new GoalFinder(FILE_AGENT);
09     initGoals(agent);
10     initThread(agent);
11   }
12   public void initGoals(Agent agent) {
13     initGeneralGoals();
14     initSpecificGoals(agent);
15   }
16   public void initGeneralGoals() { ... }
17   public abstract void initSpecificGoals(Agent agent);
18
19   /* Execution Autonomy */
20   private int maxNumberOfThreadsPerAgent = 5;
21   private ActiveObject autonomyActiveObject;
22
23   public void initThread(Agent agent) {
24     // Instantiate the ConcurrencyStrategy
25     ConcurrencyStrategy concurrencyStrategy =
26                       new ThreadPerRequestStrategy();
27     autonomyActiveObject =
28               new ActiveObject(agent.getToPerformPlans(),
29                                concurrencyStrategy);
30     autonomyActiveObject.start();
31   }
32   /* Goal Management */
33   pointcut decisionMaking(Agent agent, Stimulus msg):
34        AspectTEvents.messageReceiving(agent, msg);
35
36   after(Agent agent, Stimulus stimulus):
37             decisionMaking(agent, stimulus) {
38       makeDecision(agent, stimulus);
39   }
40   public void makeDecision(Agent agent, Stimulus stimulus){
41     // Instantiate the agent goals based on the
42     // internal and external stimulus
43     ...
44   }
45   public void instantiateReactiveGoal(Agent agent,
46                                       Message msg){
47     instantiateGeneralReactiveGoal(agent, msg);
48     instantiateSpecificReactiveGoal(agent, msg);
49   }
50   public void instantiateGeneralReactiveGoal(Agent agent,
51                                   Message msg){ ... }
52   public abstract void instantiateSpecificReactiveGoal(
53                            Agent agent, Message msg);
54   ...
55 }
```

Figura 49. Aspecto do Núcleo `Autonomy`

6.2.4.
Aspectos de Extensão

O núcleo do framework AspectT foi estendido em nosso estudo de caso por diversos aspectos de extensão. Cada um deles endereça uma característica opcional relevante de agentes heterogêneos, sendo elas: (i) Aprendizagem – essa propriedade oferece ao agente a capacidade de refinar seu conhecimento, a partir da experiência obtida como resultado de suas ações, interações com o ambiente e com outros agentes; (ii) Mobilidade – essa característica endereça o comportamento necessário do agente para trafegar entre diferentes ambientes remotos; e (iii) Colaboração – define um conjunto de papéis que oferece ao agente a habilidade de colaborar com diferentes tipos de agentes.

A implementação dos aspectos de extensão que endereçam as características de Aprendizagem, Mobilidade e Colaboração de agentes, utiliza os mecanismos de declarações inter-tipos de AspectJ para estender as classes do núcleo do framework AspectT com comportamento necessário para implementar tais características. Tal comportamento adicional (implementada por meio de novos atributos e/ou métodos), é então invocado através de pontos de corte e advices definidos também por cada aspecto de extensão. As seções seguintes apresentam os respectivos aspectos de extensão que endereçam cada uma dessas características no AspectT.

6.2.4.1.
Aprendizagem

A característica de Aprendizagem está relacionada com o comportamento do agente responsável por refinar ou obter novo conhecimento. Agentes cognitivos aprendem baseado na sua experiência como resultado de suas ações, erros, interações com o ambiente e colaborações com outros agentes [95, 105]. Diferentes técnicas de aprendizado podem ser empregadas por agentes, um protocolo de aprendizado é em geral composto pelos seguintes passos [95, 105]: (i) um evento relevante é detectado; (ii) o evento é capturado e informação relevante sobre o mesmo é recolhida para a realização do processo de aprendizado; (iii) um algoritmo de aprendizado processa a informação coletada;

(iv) a informação é armazenada e como resultado pode trazer novas conclusões; e (v) caso uma nova conclusão seja obtida, o conhecimento do agente é atualizado de forma a influenciar diretamente no seu comportamento.

A característica de Aprendizagem é endereçada no framework AspectT por um padrão de projeto o qual oferece diretrizes para o projeto e implementação de tal característica usando mecanismos de orientação a aspectos. O padrão de projeto *Learning Aspect* [49] tem como propósito modularizar o interesse de aprendizado, separando completamente a estrutura básica do agente do seu protocolo de aprendizado. A estrutura geral do padrão é apresentada na Figura 50. O padrão possui 4 participantes, sendo eles: (i) *Learning Aspect* – o qual define o protocolo de aprendizagem; (ii) *Specific Learning* – responsável por implementar a parte específica da característica de aprendizagem de um tipo ou papel de agente; (iii) *Learning Component* – implementa estratégias de aprendizado específicas; e (iv) *Knowledge Element* – oferece eventos e informações contextuais que são relevantes para o processo de aprendizagem.

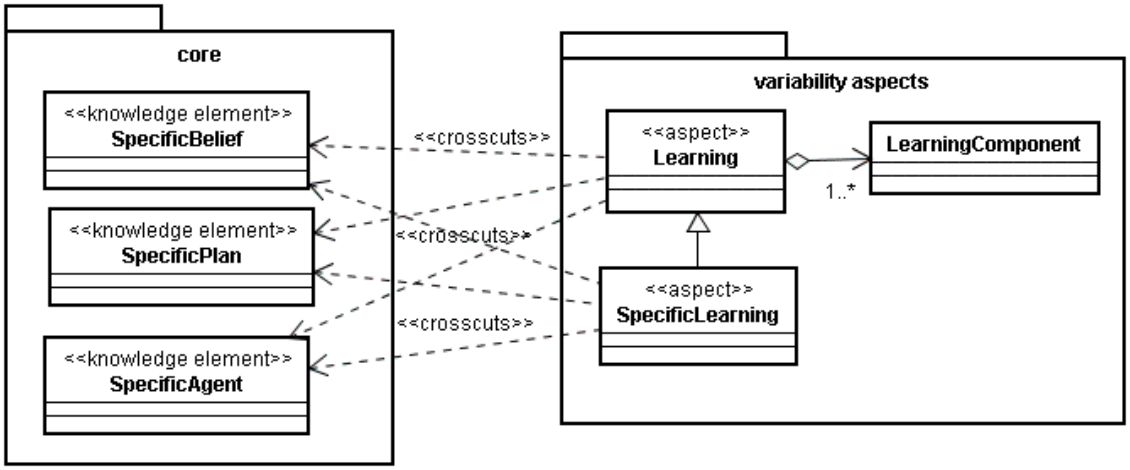

Figura 50. Estrutura do Padrão de Projeto Learning

Para ilustrar o uso do padrão de projeto *Learning Aspect* na construção de aspectos de variabilidade, apresentaremos a solução adotada na instanciação do framework AspectT para o sistema ExpertCommittee (EC). Uma das técnicas de aprendizado usadas em tal sistema para capturar as preferências dos usuários foi a *Temporal Different Learning* (TD-Learning) [95]. Os papéis de agente *Chair* e *Reviewer* do EC usam ambos a técnica de TD-Learning. O papel *Reviewer* utiliza tal técnica para capturar as preferências do usuário nos assuntos de interesse do revisor que o agente representa. Já o papel *Chair* usa tal técnica para aprender as

preferências de cada um dos revisores do sistema. É necessário a coleta de diversas informações do sistema de forma a permitir a execução efetiva de tais técnicas de aprendizado, tais como: escolhas feitas pelos usuários reais do sistema e resultado das interações entre agentes *Chair* e *Reviewer* durante avaliação de propostas de revisões de artigos. A seguir é descrita a implementação dos aspectos de aprendizagem do papel *Chair* no contexto do sistema EC.

A Figura 51 apresenta o código do aspecto abstrato `Learning`. Tal aspecto especifica: (i) um conjunto de atributos e métodos a serem introduzidos na classe plano `RevisionProposal` (linhas 7-24), os quais representam informações e comportamento úteis para implementação da característica de Aprendizagem; (ii) uma referência para uma instância da classe `TDLearning` que representa o algoritmo de aprendizado a ser utilizado pelo sistema (linha 5); e (iii) o ponto de corte abstrato `interestDegreeLearning()` – que representa pontos de execução nos quais os algoritmos de aprendizado serão executados (linhas 25-26) e que deve, portanto, ser concretizado por subaspectos de `Learning`.

```
01 public abstract aspect Learning {
02   public static final int ACCEPTED_PAPER = 200;
03   public static final int REJECTED_PAPER = 0;
04   public static final double LR = 0.1;
05   protected TDLearning learningAlgorithm;
06
07   private Hashtable RevisionProposal.proposalEvaluation =
08                            new Hashtable();
09   private int RevisionProposal.currentPaperInterest = 0;
10
11   public Hashtable RevisionProposal.getEvaluation() {
12     return proposalEvaluation;
13   }
14   public void RevisionProposal.setEvaluation(
15                                 Hashtable evaluation) {
16     this.proposalEvaluation = evaluation;
17   }
18   public int RevisionProposal.getPaperInterest() {
19     return currentPaperInterest;
20   }
21   public void RevisionProposal.setPaperInterest(
22                                           int interest) {
23     this.currentPaperInterest = interest;
24   }
25   protected abstract pointcut interestDegreeLearning(
26                 RevisionProposal proposal, Plan plan);
27 }
```

Figura 51. Aspecto de Extensão `Learning`

A Figura 52 mostra o aspecto `ChairLearning` que implementa as funcionalidades de aprendizado do papel *Chair* do EC. Ele define: (i) o ponto de corte `learningInitialization()` com um adendo associado, os quais são responsáveis pela inicialização das áreas de interesses de pesquisa de cada revisor (linhas 5-13); e (ii) uma concretização para o ponto de corte `interestDegreeLearning()` (linhas 14-18), o qual define a invocação dos algoritmos definidos pela classe `TDLearning` sempre que um agente `Chair` recebe o resultado de uma proposta de revisão (linhas 20-41). A proposta de revisão é analisada para definir se atualizações devem ser feitas nas preferências de áreas de interesse de revisores. A mudança em tais preferências, a partir da execução dos algoritmos de aprendizado, ocasiona a modificação do comportamento do sistema. Detalhes adicionais sobre os aspectos de aprendizado implementados para o EC podem ser encontradas em [46, 49].

```
01 public aspect ChairLearning extends Learning {
02   //(reviewer name, table of research interests)
03   public Hashtable reviewers = new Hashtable();
04
05   before(Agent agent, Reviewer reviewer, List papers):
06        learningInitialization(agent, reviewer, papers) {
07     String reviewerName = reviewer.getName();
08     Hashtable reviewer_interests =
09                 (Hashtable) reviewers.get(reviewerName);
10
11     // Update the reviewer interests
12     ...
13   }
14   protected pointcut interestDegreeLearning(
15               RevisionProposal proposal, Plan plan): (
16     this(plan) && args(proposal) &&
17     execution(void ProposalJudgementReceptionPlan.
18                 verifyReviewerResponse(RevisionProposal)));
19
20   after(RevisionProposal proposal, Plan plan):
21               interestDegreeLearning(proposal, plan) {
22     boolean acceptedProposal = proposal.isAccepted();
23     ResearchArea area =
24                proposal.getPaper().getResearchArea();
25     Vector paperKeywords = area.getResearchKeywords();
26
27    Hashtable reviewerEvaluation = proposal.getEvaluation();
28     int reviewerInterest = proposal.getPaperInterest();
29     Reviewer reviewer = proposal.getReviewer();
30     String reviewerName = reviewer.getName();
31
32     Hashtable reviewerPreferences =
33                (Hashtable) reviewers.get(reviewerName);
34
35     Hashtable newPreferences =
36          learningAlgorithm.learnAgentPreference(
37                   reviewerPreferences, paperKeywords,
38                   acceptedProposal, reviewerInterest);
39     learningAlgorithm.updatePreferences(
40                   reviewerPreferences,newPreferences);
41   }
42 }
```

Figura 52. Aspecto de Extensão `ChairLearning`

6.2.4.2. Mobilidade

A característica de Mobilidade está relacionada com o comportamento do agente necessário a sua movimentação para ambientes remotos. Agentes podem se mover de um ambiente/máquina em uma rede de computadores para outro(a) de forma a alcançar seus objetivos. A característica de Mobilidade envolve o endereçamento de vários interesses [116], tais como: (i) a especificação dos elementos móveis; (ii) a descrição de situações nas quais os agentes devem se mover e quando devem retornar, bem como as respectivas ações a serem tomadas antes de tais deslocamentos; e (iii) o controle dos itinerários dos agentes.

No framework AspectT, a característica de Mobilidade também foi endereçada pela elaboração de um padrão de projeto. O padrão de projeto *Mobility Aspect* [48] propõe a modularização do interesse de mobilidade de forma separada dos interesses básicos do agente e de colaboração. Ele possui 5 participantes: (i) *Mobile Element* – representa elementos do agente móvel, cujo principal propósito é modularizar outros interesses do agente, tais como um tipo ou papel de agente; (ii) *Mobility Aspect* – implementa a parte genérica do interesse de mobilidade; (iii) *Specific Mobility* – endereça a parte específica do interesse de mobilidade de determinados tipos ou papéis de agentes; e (iv) *Mobility Framework* – oferece um conjunto de serviços de mobilidade que podem ser usados na aplicação.

Para ilustrar o uso do padrão de projeto *Mobility Aspect*, vamos apresentar a instanciação do mesmo no contexto do sistema ExpertCommittee (EC). A Figura 53 apresenta a aplicação do padrão na implementação do papel *Chair* no EC. O aspecto `Mobility` afeta as classes `Agent`, `Goal` e `Plan` para introduzir comportamento de mobilidade. O subaspecto `ChairMobility` introduz o comportamento específico de mobilidade relacionado ao papel *Chair* entrecortando as classes `ResearcherUserAgent` e `InformationSearchingPlan`. Finalmente, a classe `MobilityModule` se responsabiliza pela comunicação com a plataforma JADE de forma a poder usufruir dos serviços de mobilidade oferecidos.

A Figura 54 mostra o código do aspecto abstrato de extensão `Mobility`. Ele é responsável por definir o comportamento básico do interesse de mobilidade. Ele introduz diversos atributos (linhas 4-8) e métodos (17-33) na interface

`MobileElement`, através do uso de declarações inter-tipos. Essa interface é usada para representar um elemento móvel. Subaspectos de `Mobility` definem quais tipos de agente devem implementar tal interface. Os seguintes atributos e métodos são introduzidos em tal interface: (i) atributo `home` – o qual armazena o computador original do agente (linha 4); (ii) atributo `location` – armazena a localização atual do agente (linha 5); (iii) atributo `mobilityComponent` – oferece serviços para o agente se deslocar para outros ambientes usando um framework de mobilidade específico (linha 7). No sistema EC, o componente de mobilidade foi implementado usando o framework JADE; (iv) atributo booleano `agentOut` – indica se o agente se moveu para outro ambiente diferente de sua localização original (linha 8); e (v) métodos `prepareMessageDeparture()` e `prepareMessageArrival()` – que são usados para a definição de mensagens de notificações durante a partida ou chegada, respectivamente, de um agente a um dado ambiente (linhas 17-24).

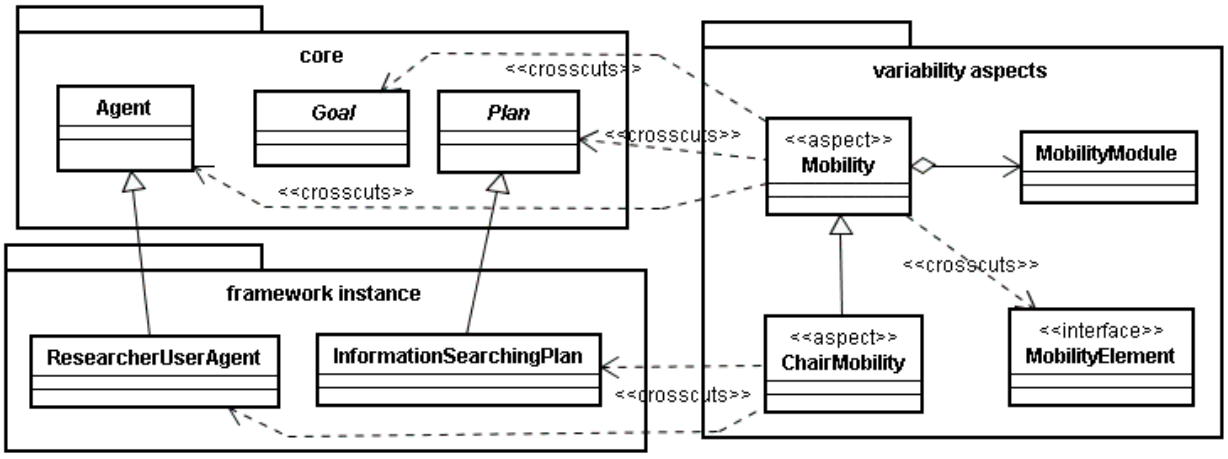

Figura 53. Aspecto de Extensão `Mobility`

Como também pode ser visto na Figura 54, o aspecto `Mobility` também define diversos pontos de corte e métodos abstratos a serem implementados por aspectos concretos de mobilidade, entre eles: (i) ponto de corte `moving()` – define pontos de junção na execução do agente que são candidatos para inicializar a sua mobilidade (linha 35); (ii) método `init()` – inicializa os serviços de mobilidade do agente em uma dada plataforma (linha 15) ; (iii) método `move()` – usado para definir as ações a serem executadas para movimentar um agente para um outro

ambiente (linha 16); (iv) método `initializeAgentInNewEnvironment()` – define a inicialização do agente após a sua movimentação para um outro ambiente (linha 44-45); e (v) método `checkMobilityNeed()` – verifica a necessidade de movimentação ou não do agente (linhas 46-47). Subaspectos de mobilidade devem concretizar tais pontos de corte e métodos de forma a endereçar a implementação do interesse de mobilidade para algum tipo ou papel de agente.

A Figura 55 apresenta o subaspecto de mobilidade `ChairMobility`. Ele é responsável por implementar o interesse de mobilidade para o papel *Chair* do EC. Agentes que assumem tal papel precisam se mover quando há uma falha na obtenção de alguma informação por algum de seus planos. Esse aspecto define, por exemplo, que o ponto de corte `moving()` deve interceptar o método `searchInformation()` da classe `InformationSearchPlan`, para decidir pela necessidade de mobilidade ou não do agente (linhas 21-23). No EC, a plataforma de mobilidade utilizada foi baseada no framework JADE. Esse framework oferece um conjunto de serviços para implementação do interesse de mobilidade do agente. A utilização de tais serviços requer a especialização da classe `Agent` do framework JADE. A classe `JADEModule` assume tal propósito, e é usada pelo aspecto `ChairMobility` para implementar as funcionalidades de mobilidade associadas aquele papel. O método `init()`, por exemplo, é usado para inicializar as funcionalidades de mobilidade do agente na plataforma JADE (linhas 8-13). Ele invoca o método `createJADEReference()` da classe `JADEModule`, para criar um agente JADE que permitirá usufruir dos serviços de mobilidade de tal plataforma (linha 10). Já o método `move()` do aspecto `ChairMobility`, invoca o método `doMove()` da classe `JADEModule` (linhas 14-20).

```
01 public abstract aspect Mobility {
02   declare parents:
03       Agent || AbstractPlan || ... implements Serializable;
04   private String MobileElement.home;
05   private String MobileElement.location;
06
07   protected JADEModule MobileElement.mobilityComponent;
08   private boolean MobileElement.agentOut;
09   protected abstract pointcut agentInstantiation(
10                                          Agent agent);
11   after(Agent agent) : agentInstantiation(agent) {
12     agent.setAgentOutFalse();
13     init(agent);
14   }
15   public abstract void init(Agent agent);
16   public abstract void move(Agent agent);
17   public Message
18       MobileElement.prepareMessageDepartureNotification(){
19     ...
20   }
21   public Message
22      MobileElement.prepareMessageArrivalNotification() {
23      ...
24   }
25   public boolean MobileElement.isAgentOut() {
26      return this.agentOut;
27   }
28   public void MobileElement.setAgentOutTrue() {
29     this.agentOut = true;
30   }
31   public void MobileElement.setAgentOutFalse() {
32     this.agentOut = false;
33   }
34   ...
35   protected abstract pointcut moving(Plan plan);
36   after(Plan plan) returning(Object result): moving(plan) {
37     Agent agent = plan.getAgent();
38     boolean moveAgent = checkMobilityNeed(agent, result);
39     if (moveAgent && (!agent.isAgentOut())) {
40       agent.prepareDepartureNotification();
41       move(agent);
42     }
43   }
44   public abstract void initializeAgentInNewEnvironment(
45                          Object mobileAgent);
46   protected abstract boolean checkMobilityNeed(Agent agent,
47                   Object result);
48   ...
49 }
```

Figura 54. Aspecto de Extensão `Mobility`

```
01 public aspect ChairMobility extends Mobility {
02   declare parents: ResearcherUserAgent implements
03                                              MobileElement;
04   declare parents: Agent || Agenda implements Serializable;
05   protected pointcut agentInstantiation(MobileElement agent):
06              this(agent) &&
07              initialization(ResearcherUserAgent+.new(..));
08   public void init(MobileElement agent) {
09     JADEModule mobilityComponent =
10            JADEModule.createJADEReference((Agent)agent);
11      agent.setMobilityModule(mobilityComponent);
12      ...
13   }
14   public void move(MobileElement agent) {
15     Location destination = new
16         jade.core.ContainerID(agent.getCurrentDestination(),
17                               null);
18     ((JADEModule)agent.getMobilityModule()).
19                                    doMove(destination);
20   }
21   protected pointcut moving(Plan plan):
22     (this(plan) && execution(Hashtable
23     InformationSearchingPlan.searchInformation(..)));
24
25   protected boolean checkMobilityNeed(MobileElement agent,
26                                       Object result) {
27    ...
28   }
29   protected pointcut afterMove(Object JADEagent):
30      this (JADEagent) &&
31      execution(* JADEModule.afterMove());
32   public void initializeAgentInNewEnvironment(Object
33     JADEagent) {
34     ...
35   }
36   pointcut informationNeedChecking(Plan plan):
37     this(plan) &&
38     execution(void PaperDistributionPlan.executePlan(..));
39
40   before(Plan plan): informationNeedChecking(plan) {
41     ...
42   }
43 }
```

Figura 55. Aspecto de Extensão `ChairMobility`

6.2.4.3. Colaboração

A característica de Colaboração possibilita a um dado agente interagir com outros agentes através do desempenho de papéis. Um papel de agente oferece capacidades extras de conhecimento, interação, adaptação ou autonomia. Durante sua execução, é comum um agente assumir diferentes papéis. Na nossa implementação, cada papel é implementado por uma interface e um aspecto. A interface especifica o comportamento do papel, através da definição de métodos. O aspecto define que a classe `Agent` ou algum de seus subtipos deverá implementar tal interface. Atributos que devem existir no papel do agente podem também ser incluídos na interface pelo aspecto, usando declarações inter-tipo. Esses atributos representam crenças específicas determinadas pelos papéis. Planos e objetivos específicos do papel podem também ser definidos. A definição de novas propriedades para um papel de agente, podem ser implementadas, através da especialização dos aspectos abstratos `Autonomy`, `Adaptation`, `Mobility` e `Learning`.

Dois papéis de agentes foram implementados para o sistema EC, são eles: `Chair` e `Reviewer`. A Figura 56 mostra o código do aspecto de extensão `Chair` do EC. Ele é responsável pela introdução de diversos atributos e métodos na classe `ResearcherUserAgent`, que são relacionados a implementação do papel `Chair` no sistema. Os seguintes atributos são definidos, entre eles: (i) um plano para a distribuição de artigos para o comitê de programa (linha 3); (ii) uma lista de artigos que foram submetidos (linha 4); uma lista de revisores de artigos (linha 5); e (iv) a data limite para submissão de artigos (linha 6). Métodos acessores para tais atributos podem também ser definidos, assim como outros que representem serviços específicos daquele papel de agente. A implementação de um papel de agente, envolve também a definição de novas classes representando objetivos, planos e crenças. Além disso, propriedades específicas de um dado papel de agente, podem também ser implementadas na forma de subaspectos dos aspectos do núcleo e extensão apresentados e exemplificados nas seções anteriores.

```
1 public aspect Chair {
2    ...
3    private DistributionPlan UserAgent.distributionPlan;
4    private List UserAgent.papersList;
5    private List UserAgent.reviewerList;
6    private GregorianCalendar UserAgent.paperSubmissionDate;
7    ...
8    public List ResearcherUserAgent.getPapersList() {
9      return this.papersList;
10   }
11   public List ResearcherUserAgent.getReviewerList() {
12     return this.reviewerList;
13   }
14   public void ResearcherUserAgent.setPapersList(
15                                        List papersList) {
16     this.papersList = papersList;
17   }
18   public void ResearcherUserAgent.requestInformation(){
19     ...
20   }
21 }
```

Figura 56. Aspecto de Extensão de Colaboração `Chair`

6.2.5. Modelo Generativo

O modelo generativo do framework AspectT é apresentado na Figura 57. O modelo de arquitetura contempla os seguintes componentes principais: (i) core – o qual agrega todas as classes e aspectos do núcleo do framework na forma de um conjunto de subcomponentes relacionados com cada uma das características do framework, tais como, knowledge, interaction, adaptation, autonomy e ejps. O componente ejps é responsável por agregar o aspecto que expõe os pontos de junção de extensão definidos pelo AspectT; e (ii) extensions - agrega aspectos e classes relacionadas com a implementação de extensões ao framework AspectT, é organizado nos seguintes subcomponentes: learning, mobility e collaboration. Cada um dos componentes do modelo de arquitetura do AspectT também agrega um conjunto de templates que, em geral, representam: (i) especializações de classes ou aspectos abstratos do framework (exemplos: templates `UserAgentTemplate`, `InteractionTemplate` e `AdaptationTemplate`); e (ii) arquivos de configuração do sistema, tais como, um arquivo que especifica a adaptação de planos (template `PlanAdaptationXMLFile`) – determinando quais objetivos demandam a execução de quais planos), e um arquivo que especifica a

automonia reativa (template `ReactiveAutonomyXMLFile`) – o qual determina quais objetivos devem ser instanciados a partir do recebimento de determinadas mensagens.

O modelo de característica do AspectT permite a definição de vários agentes (característica Agent). Cada agente pode definir suas características e propriedades obrigatórias, tais como: (i) Knowledge – onde podem ser definidos crenças, objetivos e planos do agente; (ii) Interaction – essa característica permite especificar os sensores e efetuadores a serem usados pelo agente, assim como mensagens específicas que ele pode processar; (iii) Adaptation – permite a associação de planos a determinados objetivos (Plan Adaptation) e a atualização de crenças a partir do recebimento de determinadas mensagens (Belief Adaptation); e (iv) Autonomy – permite a definição de instanciação de objetivos a partir do recebimento de determinadas mensagens (Reactive Autonomy), assim como a escolha da estratégia de concorrência a ser usada no processamento dos planos (Execution Autonomy). Para cada agente pode também ser definido um conjunto de características opcionais, tais como, Learning, Mobility e Roles. As duas primeiras são especificadas para permitir a criação dos aspectos de extensão responsáveis pela sua implementação. A característica Role representa a especificação dos papéis de agente. Ela permite especificar características de Knowledge, Interaction, Adaptation e Autonomy, Learning e Mobility específicas do papel do agente.

O modelo de configuração do AspectT define uma série de relações de dependência entre os elementos de implementação e características. A implementação de várias das características transversais de agentes usando aspectos, facilita a criação de tais relações de dependência, contribuindo para uma correspondência direta entre características e elementos de implementação. Cada um dos templates que representam especializações de classes ou aspectos abstratos do framework é diretamente relacionado com a característica responsável por coletar informações para a sua instanciação. Exemplos são: `PlanTemplate` relacionado com a característica Plan; `AutonomyTemplate` relacionado com a característica Autonomy, e assim por diante. Em seu estágio atual, o modelo de configuração do AspectT apenas define relações de dependência entre elementos de implementação e características. Relações tranversais válidas e mapeamentos entre características `<<joinpoint>>` e EJPs

não foram necessárias nesse estudo de caso, porque a maioria dos aspectos possui pontos de corte fixos, e, portanto pré-definidos. Em trabalhos futuros pretendemos explorar a extesão do modelo generativo do AspectT para endereçar variabilidades em pontos de corte de aspectos representando as características Learning e Mobility, principalmente.

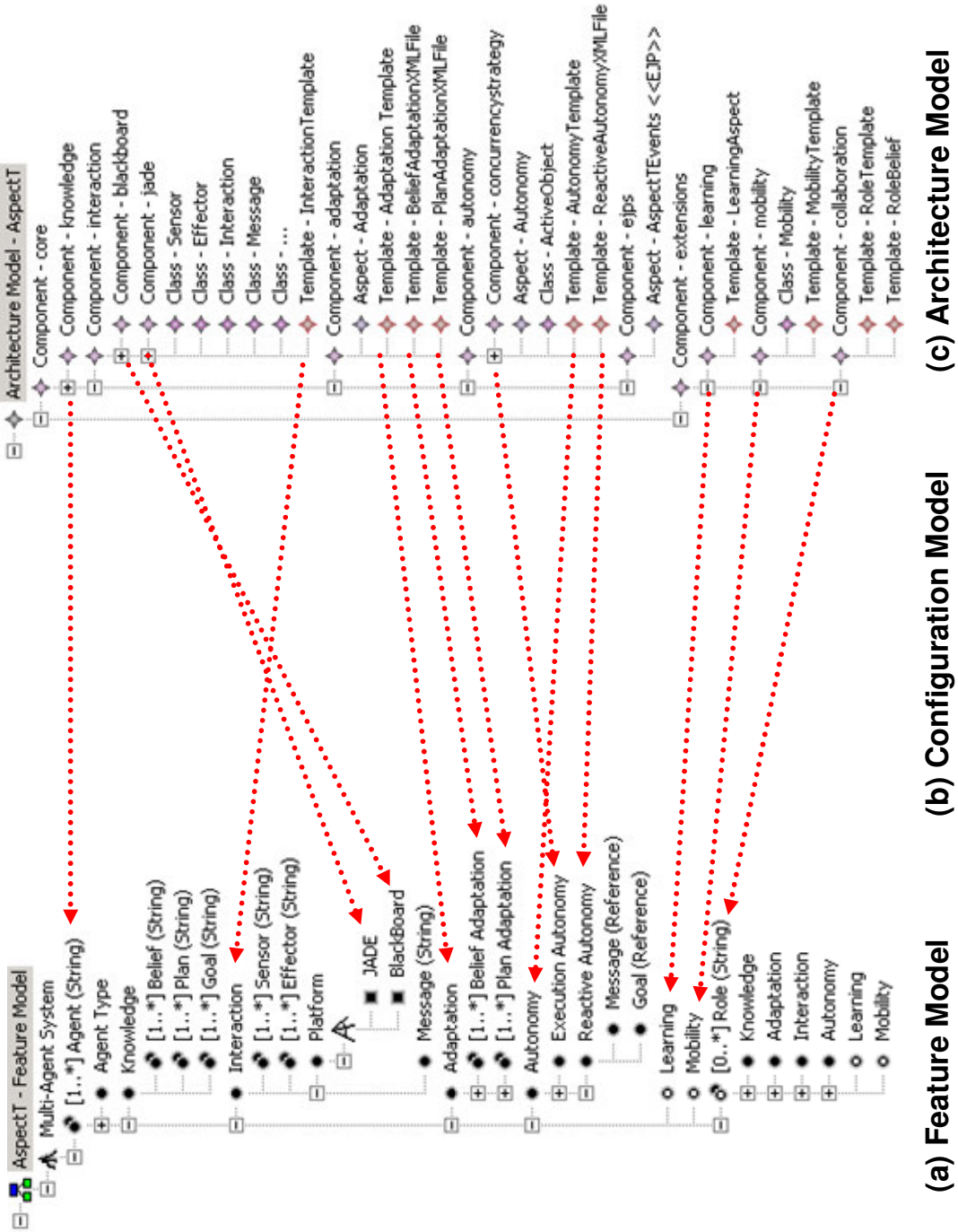

Figura 57. Modelo Generativo do AspectT

6.3. Linha de Produto de Jogos J2ME

Java 2 Micro Edition[18] (J2ME) é uma plataforma de tecnologias para o desenvolvimento de aplicações para dispositivos móveis. Nos últimos anos, J2ME tem sido usada para o desenvolvimento de jogos para celulares com complexidade considerável. Nossa abordagem também foi usada em um estudo de caso de uma linha de produto de jogos J2ME, da empresa *Meantime Mobile Creations*. O estudo de caso envolveu a refatoração da linha de produto do jogo *Rain of Fire*, originalmente desenvolvida em J2ME. Tal estudo foi conduzido por membros do *Software Productivity Group* da Universidade Federal de Pernambuco (UFPE), que participaram diretamente no desenvolvimento e evolução da abordagem baseada em EJPs.

A adaptação de jogos desenvolvidos usando a tecnologia J2ME para diferentes dispositivos portáteis, ocasiona o surgimento de várias variabilidades [3]. Essas variabilidades surgem, principalmente, em função de características do dispositivo no qual o jogo será instalado, tais como, restrições de hardware oferecidas e bibliotecas proprietárias. Exemplos de variabilidades [3, 5] para esse domínio de jogos são: características de interface gráfica (tais como, tamanho de tela, número de cores, som), memória disponível, tamanho da aplicação e bibliotecas proprietárias para manipulação de imagens. Assim, cada jogo construído para a plataforma J2ME pode ser visto como uma linha de produto em função da sua adaptação para diferentes dispositivos.

6.3.1. Núcleo do Framework

A estrutura e comportamento de linhas de produto de jogos é tipicamente definida por um framework conhecido nesse domínio como *Game Engine*. Um *Game Engine* pode ser caracterizado como uma máquina de estados cujas transições são definidas em função do tempo transcorrido do jogo e interações do usuário através do teclado. As mudanças de estado afetam objetos visuais do jogo

[18] Sun Microsystems. Java 2 Platform, Micro Edition (J2me). URL: http://java.sun.com/j2me/. 2004.

(tais como, atores e ambiente) e a forma como eles interagem. Dessa forma, cada mudança de estado demanda modificações visuais nos objetos sendo representados na tela do jogo. Pontos de extensão do framework incluem tipicamente classes abstratas que definem operações básicas para desenho de atores do jogo. A Figura 58 apresenta as classes principais do núcleo do framework do jogo *Rain of Fire*. A Figura 59 mostra como essas classes colaboram para implementar o fluxo definido para o *game engine*.

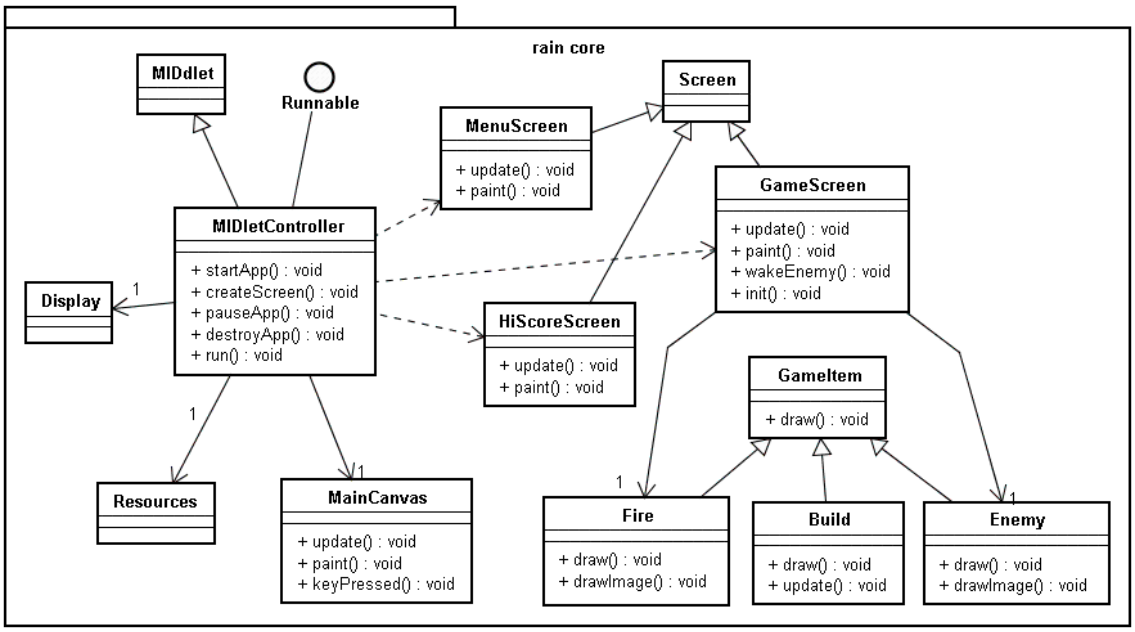

Figura 58. Diagrama de Classes do núcleo do *Rain of Fire*

6.3.2. Pontos de Junção de Extensão

O *Game Engine* que define o núcleo do framework da linha de produto deve definir pontos de junção de extensão (EJPs) para permitir a composição de extensões tranversais na sua funcionalidade básica. Os seguintes EJPs foram definidos: (i) operações de inicialização e uso de imagens do jogo; (ii) operações de desenho de imagens específicas; e (iii) eventos de inicialização e de mudanças de tela do jogo. Esses EJPs foram definidos porque representam eventos relevantes do fluxo de execução do *game engine*. A Figura 59 mostra alguns dos EJPs expostos pelo *game engine* dentro do fluxo principal de colaboração de suas classes. A Figura 60 apresenta a arquitetura geral do jogo *Rain of Fire* com seus

respectivos aspectos EJPs e aspectos de extensão. A seção seguinte descreve como vários dos EJPs são usados pelos aspectos de extensão para estender o comportamento do *game engine*.

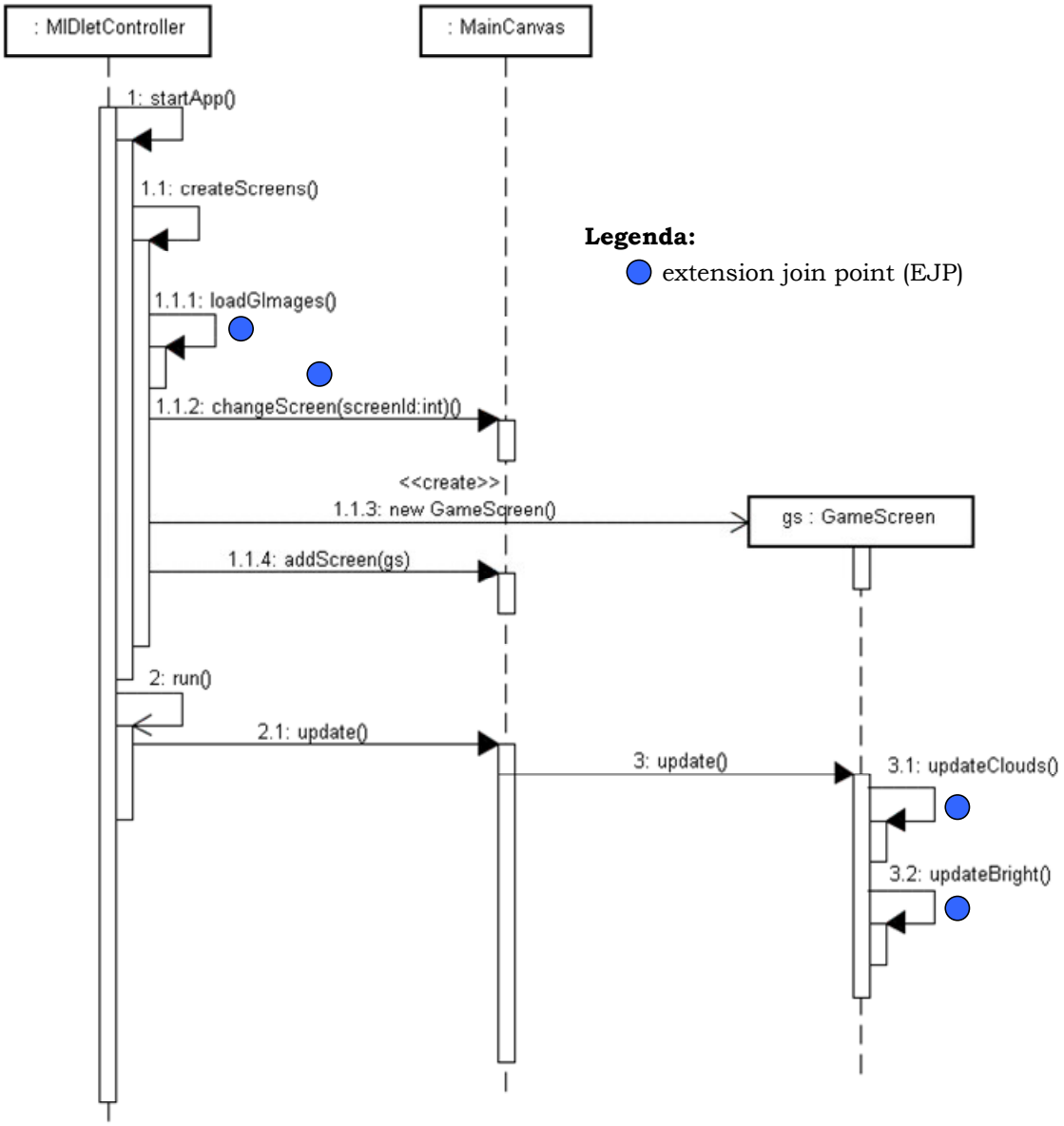

Figura 59. Diagrama de Seqüência do *Rain of Fire*

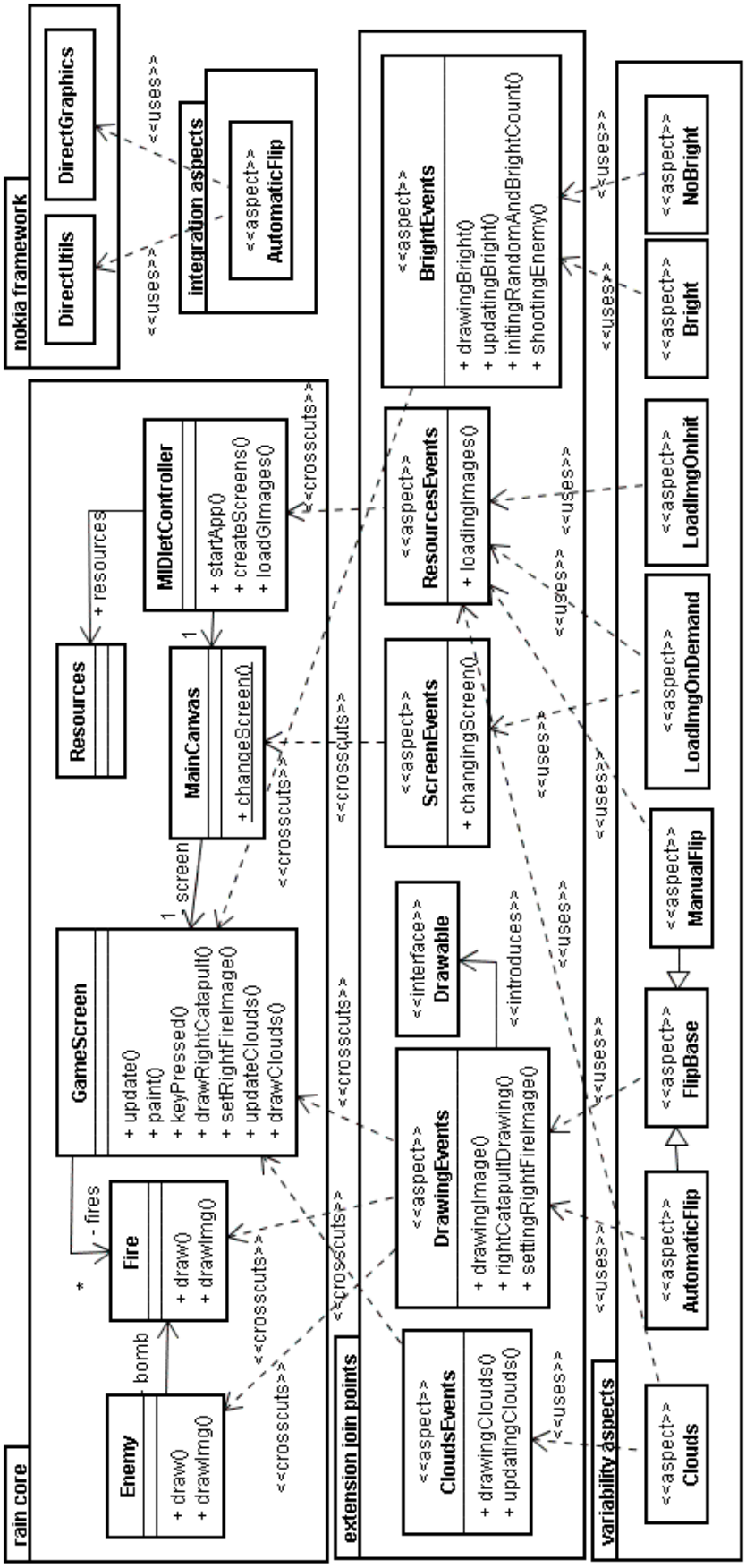

Figura 60. Arquitetura da Linha de Produto *Rain of Fire*

6.3.3. Aspectos de Extensão

Diversos aspectos de extensão foram implementados para endereçar características transversais opcionais e alternativas existentes na linha de produto. Esses aspectos representam a implementação de variabilidades presentes em jogos J2ME. A Figura 60 mostra o núcleo do framework, com seus respectivos EJPs e aspectos de extensão.

Aspectos de variabilidade foram definidos para implementar a característica opcional Croma que representa imagens decorativas presentes no cenário de um jogo. Um exemplo é a existência de imagens representando nuvens que passam no fundo de tela de determinados cenários do jogo. A característica Croma é considerada opcional na linha de produto, porque dispositivos com baixa capacidade de recursos podem não incluir tais características. O aspecto de variabilidade `Clouds` representa a implementação da característica Croma para a linha de produtos do jogo *Rain of Fire*. Observe na Figura 60 que esse aspecto faz uso dos EJPs `CloudsEvents` e `ResourceEvents` para inserir tal funcionalidade no núcleo do framework que define o jogo. A Figura 61 mostra o código parcial do aspecto `Clouds`. A implementação dele envolve: (i) declarar atributos para carregar tais arquivos de imagens (linhas 2-7); (ii) carregar arquivos de imagens (linhas 15-25); e (iii) desenhar e atualizar tais imagens em função de mudanças no estado do jogo (linhas 8-14 e linhas 26-40). O código responsável pelo desenho dessas imagens decorativas é encontrado em diferentes trechos de código de várias classes. Dessa forma, ele pode ser visto como sendo uma característica transversal.

Aspectos de variabilidade e integração foram também codificados para implementar características alternativas de desenho de imagens. Imagens específicas do jogo podem ser desenhadas em vários trechos de código e, em algumas circunstâncias, podem ser transformadas (rotacionadas, flipped), manualmente usando novas imagens (aspecto `ManualFlip`) ou manipulando as imagens originais através do uso de bibliotecas proprietárias (aspecto `AutomaticFlip`). O aspecto `AutomaticFlip` é considerado de integração porque ele envolve interação com bibliotecas proprietárias. A Figura 60 apresenta tais aspectos e os respectivos EJPs `DrawingEvents` e `ResourcesEvents` que garantem

uma implementação modular para a implementação de suas respectivas características.

```
01 public privileged aspect Clouds {
02   public static Image clouds01 = null;
03   public static Image clouds02 = null;
04   ...
05   public int clouds01_x;
06   public int clouds02_x;
07   ...
08   void around(GameScreen gs):
09            CloudsEvents.updatingClouds(gs) {
10     updateClouds(gs);
11   }
12   void around(Graphics g): CloudsEvents.drawingClouds(g) {
13     drawClouds(g);
14   }
15   before(): ResourcesEvents.loadingImages() {
16     loadImages();
17   }
18   public void loadImages() {
19     try {
20       clouds01 = Image.createImage(... + "clouds01.png");
21       ...
22     } catch(IOException ioe) {
23       ...
24     }
25   }
26   protected void updateClouds(GameScreen gs) {
27     if (clouds01_x <= -clouds01.getWidth()&&gs.aux==255){
28       clouds01_x = Resources.CANVAS_WIDTH;
29     }
30     ...
31     // Updates clouds movement
32     if (MainCanvas.frame % 8 == 0) {
33       clouds01_x -= 1;
34     }
35     ...
36   }
37   protected void drawClouds(Graphics g) {
38     g.drawImage(clouds01, clouds01_x, 87, g.TOP | g.LEFT);
39     ...
40   }
41 }
```

Figura 61. Aspecto de Variabilidade `Clouds`

Finalmente, aspectos de variabilidade foram também definidos para implementar a característica opcional de otimização. Essa característica envolve a carga de imagens por demanda quando alternando entre telas do jogo. Essa política de carga é usada para dispositivos que possuem restrições nos recursos disponíveis (ex: memória). Dispositivos sem tais restrições não necessitam da implementação de tal característica, e seu código carrega todas as imagens do jogo

durante a inicialização do mesmo. A Figura 60 mostra os aspectos `LoadImgOnDemand` e `LoadImgOnInit` que realizam a carga de imagens, respectivamente, por demanda ou na inicialização. Os EJPs `ScreenEvents` e `ResorcesEvents` são usados por tais aspectos para permitir a extensão do núcleo do framework. Detalhes adicionais sobre a implementação desse estudo de caso podem ser encontradas em [5, 73].

6.3.4. Modelo Generativo

O modelo generativo da linha de produtos do jogo J2ME *Rain of Fire* é apresentado na Figura 62. Ele apresenta o modelo de arquitetura agregando os elementos de implementação do jogo em 2 componentes principais: core e extensions. O componente core agrega as classes que definem o comportamento básico do jogo, assim como os aspectos EJPs (`ResourceEvents` e `DrawingEvents`) que são compartilhados por vários dos aspectos de variabilidade. O componente extensions agrega vários sub-componentes cada um deles representando as variabilidades da linha de produto, sendo eles: bright, flip, clouds e imageloading. Observe que cada um desses componentes agrega os respectivos aspectos de variabilidade associados. Os componentes bright e clouds agregam ainda os aspectos EJPs (`BrightEvents` e `CloudEvents`, respectivamente) que são associados apenas aquelas variabilidades.

O modelo de características do jogo *Rain of Fire* apresenta apenas as variabilidades da linha de produto, contendo: (i) 2 características opcionais (Bright e Clouds) e (ii) 2 características alternativas (Flip e Image Loading). Todas essas características são `<<crosscutting>>` porque representam a implementação de aspectos de extensão. Nesse estudo de caso não foi necessário representar nenhuma característica `<<join point>>`, porque todos os aspectos do modelo de característica possuem pontos de corte fixos. Como conseqüência, o modelo de configuração desse estudo de caso não necessita definir: (i) relações transversais válidas entre características `<<crosscutting>>` e `<<join point>>`; e (ii) o mapeamento entre características `<<join point>>` e pontos de junção concretos.

O modelo de configuração desse estudo de caso se resume, portanto, as relações de dependência entre elementos de implementação e características. A Figura 62 apresenta tais relações de dependência. O aspecto `Bright` depende da seleção da característica Bright. Os aspectos do componente flip dependem, respectivamente, de cada uma das alternativas (Automatic, Manual) da característica Flip. O mesmo ocorre para os aspectos do componente imageloading que dependem das alternativas oferecidas para as características Image Loading. Dois elementos de implementação dependem da da característica alternativa On Demand: (i) o aspecto `LoadImgOnDemand` e (ii) o aspecto EJP `ScreenEvents` usado pelo primeiro.

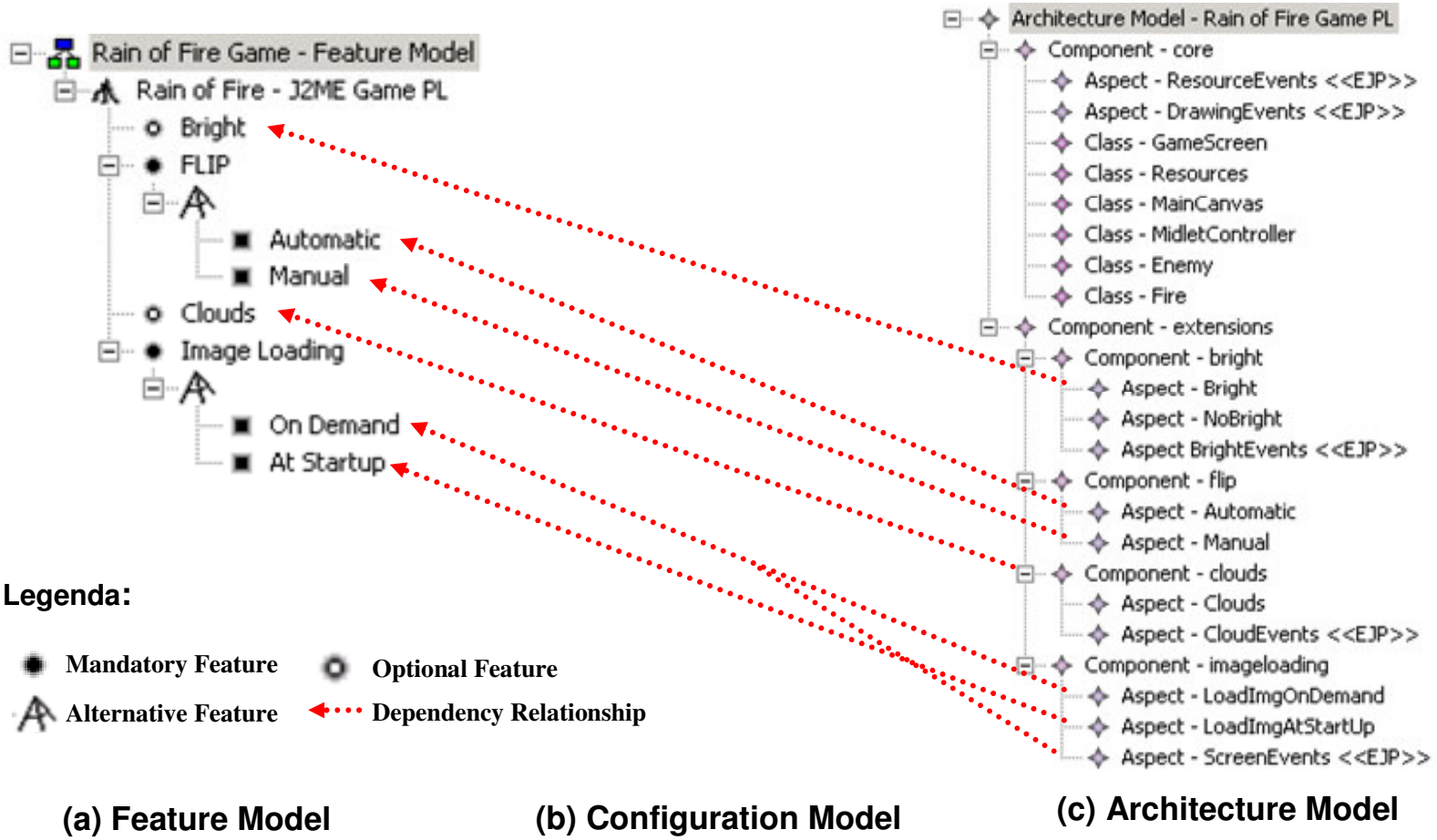

Figura 62. Modelo de Configuração da Linha de Produto do Rain of Fire

6.4. Sumário

Este capítulo apresentou, em detalhe, três diferentes estudos de caso de frameworks desenvolvidos com o uso da nossa abordagem. Os estudos envolveram frameworks de três diferentes domínios (*measurement*, sistemas multi-agentes, jogos para celulares) demonstrando a generalidade da abordagem proposta. Os pontos de junção de extensão (EJPs) encontrados para cada estudo de caso representam eventos ou estados relevantes que ocorrem no domínio do framework. Diferentes tipos de aspectos do núcleo e de extensão foram também definidos para cada um dos frameworks. No capítulo seguinte são discutidos os benefícios observados assim como lições aprendidas a partir da realização desses estudos de caso.