4 MAS-School - Um Método para Incluir Aprendizado em Sistemas Multi-Agente

A tecnologia de agentes é utilizada em muitos simuladores e sistemas inteligentes para auxiliar as pessoas em várias tarefas complexas. Aplicações para o Comércio Eletrônico, Extração de Informação, e *Business Intelligence* também estão utilizando esta tecnologia para resolver problemas complexos de forma assíncrona e em ambientes distribuídos. Neste contexto, os algoritmos de *Machine Learning* ou Aprendizado de Máquina são cruciais para fornecer estratégias conhecidas para construir agentes em ambiente abertos e heterogêneos como a Internet.

Porém, incluir essas técnicas de inteligência em sistemas multi-agente de larga escala não é uma tarefa simples. O *design* e implementação de tais sistemas inteligentes sempre apresentam questões semelhantes, dentre as quais:

(viii) Como avaliar o objetivo do sistema como um todo?

(ix) Como definir e avaliar o objetivo individual de cada agente?

(x) Como modelar o conhecimento de cada agente?

(xi) Como projetar o mecanismo de aquisição de conhecimento para cada agente?

(xii) Como combinar múltiplas técnicas de aprendizado e distribuir essas técnicas para cada agente no sistema?

(xiii) Como associar abstrações de agentes com abstrações de *machine learning*?

(xiv) Como especificar abstrações de *machine learning* nas primeiras fases de *design* e permitir uma transição para a fase de implementação?

Infelizmente, os engenheiros de software ainda utilizam a sua experiência e intuição para resolver as questões acima. Muita pesquisa tem sido feita para criar metodologias e *frameworks* de implementação para sistemas multi-agente. Porém, nenhum desses trabalhos apresenta um guia para incluir técnicas de aprendizado já na fase inicial de *design*. Os *frameworks* (Howden et al., 2001; Telecom Itália-

Jade, 2003) de implementação disponibilizam APIs para desenvolver sistemas multi-agente, mas não orientam a estruturação do *design* do aprendizado de uma maneira sistemática. Além disso, muitas metodologias (Zambonelli et al., 2003; Deloach, 1999; Bresciani et al., 2004; Padgham et al., 2002) de desenvolvimento orientadas a agentes são focados em um nível muito alto de abstração, e não indicam como tratar aprendizado desde a fase de *design* até a implementação.

Este capítulo apresenta o método MAS-School (Sardinha et al., 2004b; Sardinha et al., 2005b) para incluir técnicas de *machine learning* desde as primeiras fases de *design*. Esse método apresenta várias orientações de como incluir aprendizado na fase de *design* e implementação. O método apresenta no final uma estratégia incremental de desenvolvimento para permitir a avaliação das técnicas de *machine learning*.

4.1. Incluindo Aprendizado em Sistemas Multi-Agente

Mitchell (Mitchell, 1997) define *machine learning* como : "Diz-se que um programa de computador aprende a partir de uma experiência *E* em relação a uma classe de tarefas *T* e medida de performance *P*, se a sua performance nas tarefas *T*, medida por *P*, melhora com a experiência *E*". Conseqüentemente, as técnicas de aprendizado são normalmente utilizadas para melhorar a performance de um sistema.

O principal objetivo do método aqui proposto é incluir disciplinadamente as técnicas de *machine learning* em sistemas multi-agente. Esse método também permite, em sua fase de implementação, a integração de diferentes algoritmos de *machine learning* e a avaliação da performance do sistema. O método possui quatro fases distintas:

(i) *Objetivo Sistêmico & Seleção da Medida de Performance*, onde um objetivo e uma medida de performance são selecionados para o sistema;

(ii) *Seleção do Agente & Definição do Objetivo do Aprendizado no Agente*, onde agentes são selecionados, e objetivos são atribuídos para o algoritmo de aprendizado;

(iii) *Design do Aprendizado no Agente*, onde o *design* de código é definido; e

(iv) *Implementação Incremental & Avaliação de Performance*, onde uma implementação incremental é proposta com treinamento, teste e avaliação.

A figura 29 apresenta as quatro fases do método. A seguir apresentamos em detalhe cada fase do método.

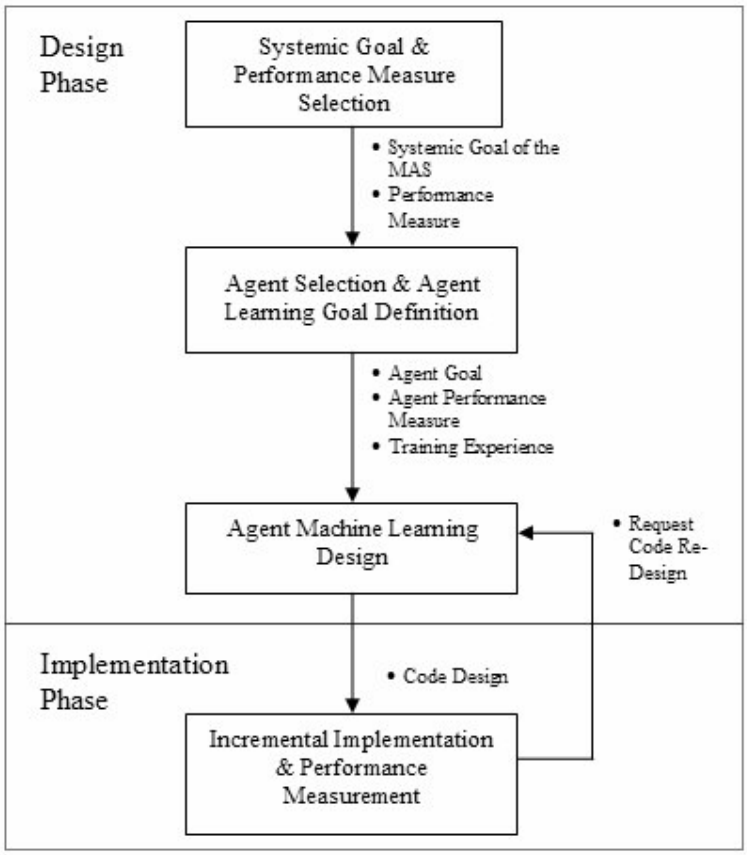

Figura 29: O método para incluir Aprendizado em um Sistema Multi-Agente.

4.1.1. Objetivo Sistêmico & Seleção da Medida de Performance

O Engenheiro de Software deve definir nesta fase dois elementos centrais que estão relacionados com aprendizado: (i) *Objetivo Sistêmico*, SG, e (ii) *Medida de Performance Sistêmica*, SP, que mede o ganho de performance do sistema. O *Objetivo Sistêmico* é o objetivo de mais alto nível do sistema, e é definido nas

primeiras fases de modelagem de um sistema. A *Medida de Performance Sistêmica* é o mecanismo para avaliar se o objetivo está sendo atingido. Conseqüentemente, a *Medida de Performance* deriva diretamente do *Objetivo Sistêmico*. Por exemplo, na técnica orientada a objetivos onde se utiliza o processo recursivo de decomposição de um objetivo do problema principal em vários objetivos de subproblemas, o *Objetivo Sistêmico*, SG, é igual ao Objetivo do Problema Principal conforme ilustra a figura 30. A *Medida de Performance Sistêmica*, SP, é normalmente uma variável numérica que permite avaliar se o objetivo sistêmico está sendo atingido. Por exemplo, se o sistema multi-agente é responsável por gerenciar uma fábrica, a *Medida de Performance Sistêmica*, SP, pode ser o lucro gerado pela fábrica.

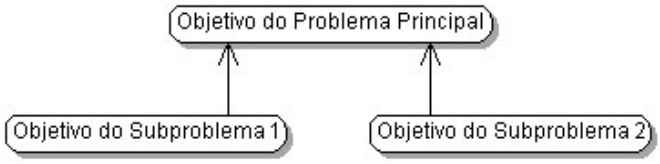

Figura 30: A decomposição dos objetivos relacionados com o comércio de bens.

A abstração de objetivos é central neste método, pois permite a definição de uma medida de performance para cada agente na seção 4.1.2, e a definição do *design* das técnicas de *machine learning* na seção 4.1.3. Conseqüentemente, o processo de decomposição dos objetivos em sub-objetivos é fundamental para este processo. É importante frisar que essa modelagem de hierarquia de objetivos é uma atividade comum nas metodologias orientadas a agentes. Entretanto, o principal objetivo é detectar e modelar objetivos específicos de aprendizado. Os objetivos funcionam como uma abstração para unificar conceitos de aprendizado com conceitos básicos de sistemas multi-agente.

No processo de modelagem, os tipos de agentes são criados para conduzir os objetivos da figura 30. A tabela 7 apresenta um exemplo de mapeamento entre esses objetivos e tipos de agentes. Na próxima fase do método, alguns desses agentes são selecionados para utilizar técnicas de *machine learning*.

Objetivo	Agente
Objetivo do Subproblema 1	Agente I
Objetivo do Subproblema 2	Agente II

Tabela 7: A mapeamento entre subproblemas e tipos de agentes.

4.1.2. Seleção do Agente & Definição do Objetivo do Aprendizado no Agente

Essa fase seleciona os agentes definidos na tabela 7 que possuem planos com alta probabilidade de melhorar a *Medida de Performance Sistêmica*, SP. Conseqüentemente, esses planos precisam utilizar técnicas de *machine learning* para melhorar a performance do sistema. O objetivo é estabelecer um *Problema de Aprendizado do Agente* bem definido. Conseqüentemente, três características são definidas: (i) *Objetivo do Aprendizado*, G; (ii) *Medida de Performance*, P, que mede a melhora da performance no agente individualmente; e, (iii) uma *Experiência de Treinamento*, E, que define o processo de aquisição do conhecimento no agente com o aprendizado.

Por exemplo, o Agente I da tabela 7 pode ser um preditor de preços de leilões de hotéis, e quanto melhor for a sua predição melhor será o desempenho do sistema como um todo. Assim, esse agente deve ser selecionado para utilizar uma técnica de *machine learning*. O *Problema de Aprendizado do Agente* para o Preditor (Agente I) pode ser definido por:

(i) G: Prever os preços de leilões de hotéis;
(ii) P: O erro entre o preço previsto e o preço real; e
(iii) E: Utilizar um histórico de preços dos leilões.

4.1.3. Design do Aprendizado no Agente

Um bom *design* do agente permite a inclusão ou reuso de várias técnicas de *machine learning* e uma boa manutenção de código. O processo de maximizar a performance do agente, normalmente, precisa de vários algoritmos diferentes para

que se possa escolher o de melhor performance. A figura 31 apresenta o digrama de Classes (UML) do Preditor. Esse *design* utiliza um padrão de projeto orientado a objetos para incluir *machine learning* em agentes de software (Sardinha et al., 2004d). O *design* permite o teste e avaliação de vários algoritmos diferentes. O padrão de projeto será apresentado em detalhes na seção 4.2.

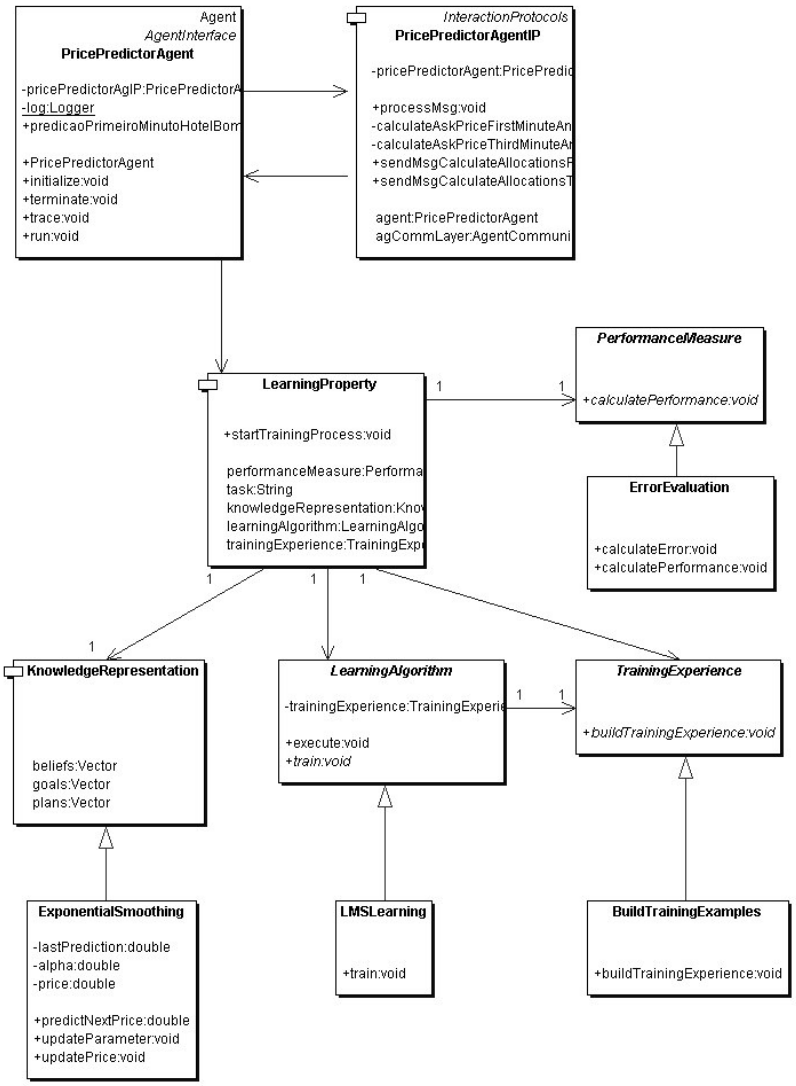

Figura 31: O diagrama de Classes (UML) do Preditor de Preços.

As classes *PricePredictorAgent* e *PricePredictorAgentIP* possuem o código dos serviços básicos do agent, tais como: tratamento de eventos, tratamento de mensagens, etc.

A classe *KnowledgeRepresentattion* é um classe abstrata da estrutura de dados do conhecimento do agente. O monitoramento da performance do agente é codificado como uma classe abstrata chamada *PerformanceMeasure*. O algoritmo de *machine learning* é uma classe abstrata chamada *LearningAlgorithm*. O gerador de exemplos definido pela "experiência de treinamento" é modelado como uma classe abstrata chamada *TrainingExperience*.

As classes concretas *ExponentialSmoothing*, *ErrorEvaluation*, *LMSLearning*, e *BuildTrainingExamples* são classes que respectivamente implementam as classes abstratas *KnowledgeRepresentation*, *PerformanceMeasure*, *LearningAlgorithm*, e *TrainingExperience*.

Vários eventos podem disparar o processo de aprendizado (Mitchell, 1997), tais como: execução de uma ação interna do agente, uma exceção levantada, a troca de mensagens entre agentes, e eventos gerados pelo ambiente externo. As classes concretas *PricePredictorAgent* e *PricePredictorAgentIP* utilizam a classe *LearningProperty* para disparar esse processo de aprendizado.

4.1.4. Implementação Incremental & Avaliação de Performance

As decisões mais importantes na fase de implementação de uma técnica de *machine learning*, nos agentes selecionados na seção 4.1.2, são: (i) a representação do conhecimento; (ii) o algoritmo de aprendizado; (iii) o conjunto de treinamento utilizado pelo algoritmo de aprendizado.

A primeira decisão determina exatamente o tipo de conhecimento aprendido. Esse conhecimento pode ser modelado como uma função *F* que recebe um estado *S* e determina uma ação *A*, ou $F: S \rightarrow A$. Porém, aprender esse conhecimento é muitas vezes uma tarefa muito difícil. Normalmente, a complexidade desse conhecimento é reduzida para aprender apenas uma representação aproximada do conhecimento, a aproximação da função *F*. Essa representação do conhecimento aproximada pode ser uma função linear com pesos, uma coleção de regras, uma rede neural, ou uma função polinomial quadrática. Essa decisão tem prós e contras, pois uma representação do conhecimento aproximado "expressiva", muito próxima da função *F*, requer um conjunto de treinamento grande na fase de treinamento.

O conhecimento do Preditor de Preços foi modelado como uma função chamada *NextPrice*. Essa função recebe preços correntes *A* e gera uma preço futuro *N* (*NextPrice*: $A \rightarrow N$). A representação aproximada da função *NextPrice* utiliza uma função para calcular o preço futuro: $PredictedPrice(n+1) = \alpha * Price(n) + (1 - \alpha) * PredictedPrice(n)$, onde α é um numero entre 0 e 1; e *n* é o *n*-ésimo instante. Esta formula é codificada na classe *ExponentialSmoothing* da figura 31. O algoritmo de aprendizado Least Mean Squares (LMS) é utilizado para adaptar o α: $\alpha(n) = \alpha(n-1) + \beta * (Price(n-1) - PredictedPrice(n-1))$, onde β é a taxa de aprendizado. Esse algoritmo é codificado na classe *LMSLearning* da figura 31.

Um conjunto de treinamento é preciso para que o agente aprenda, e a seleção de um processo para criar esse conjunto é uma decisão muito importante. O conjunto de treinamento pode ser obtido através de uma experiência direta ou indireta. Na experiência direta, o engenheiro deve cuidadosamente selecionar o melhor conjunto de treinamento que leve ao bom desempenho do conhecimento aproximado. Na experiência indireta, o conhecimento aproximado deve sugerir ações que levem a estados conhecidos que melhorem a performance do agente. Porém, esse conhecimento aproximado deve também sugerir estados desconhecidos para que novas experiências sejam adquiridas. No longo prazo, a exploração é muito importante para agentes que utilizam a experiência indireta. A classe *BuildTrainingExamples* da figura 31 codifica a consulta ao banco de dados com preços de leilões que o agente já participou. A classe *ErrorEvaluation* da figura 31 implementa a avaliação da performance definida na seção 4.1.2.

4.1.4.1. Desenvolvimento Incremental, Teste e Integração de Agentes Inteligentes

Ao invés de desenvolver o sistema multi-agente com todos os agentes inteligentes em uma única fase, esta seção apresenta uma proposta incremental para facilitar a integração e análise da performance dos agentes de software na fase de implementação. A primeira versão do sistema multi-agente deve possuir apenas de agentes simples ou reativos (Ferber, 1999) sem nenhuma técnica de *machine learning* implementada. Essa primeira versão é importante para testar a comunicação entre os agentes e a interação com o ambiente externo. A figura 32 apresenta o processo de integração dos agentes selecionados na seção 4.1.2. O

processo deve ser usado para melhorar a *Medida de Performance Sistêmica* definida na seção 4.1.1.

O desenvolvimento incremental começa com a remoção do código de um dos agentes selecionados na seção 4.1.2. O código do agente deve ser implementado usando o *design* estabelecido na seção 4.1.3, e a fase de treinamento deve vir logo em seguida, especialmente se foi escolhido a experiência direta para o processo de criação do conjunto de treinamento. Antes de reintegrar o agente no sistema, um teste individual é feito para testar a melhora de performance usando a *Medida de Performance* definida na seção 4.1.2. Este teste deve ser elaborado pelo engenheiro de sistema, e normalmente são desenvolvidos pequenos módulos de software para gerar casos de testes. Erros de codificação dos agentes "inteligentes" são encontrados nesta etapa.

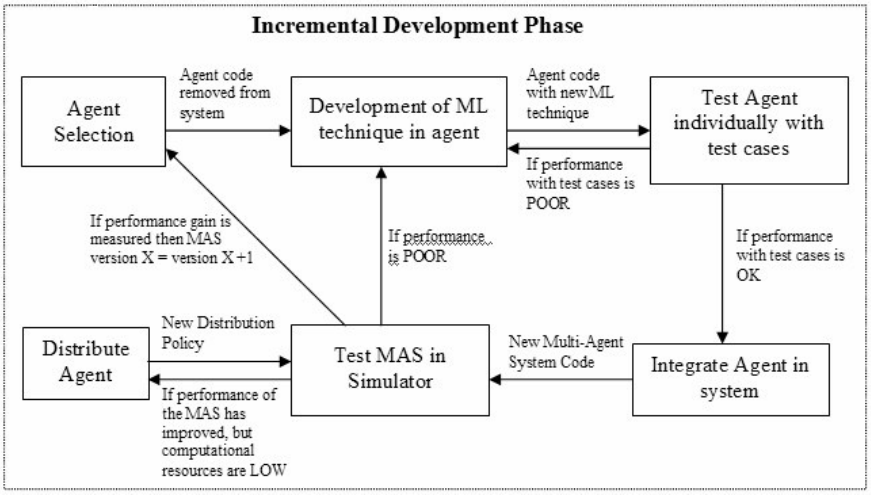

Figura 32: O processo incremental de desenvolvimento.

A reintegração do agente no sistema é feita depois que os casos de testes confirmam que o algoritmo de *machine learning* está funcionando. Esses casos de testes não são capazes de garantir a melhora na *Medida de Performance Sistêmica* definida na seção 4.1.1. Conseqüentemente, um teste de performance deve ser feito com o sistema multi-agente e novo agente inteligente. É preciso ter um simulador do ambiente para esta fase, e uma nova versão é dada para o sistema caso a *Medida de Performance Sistêmica* indicar que houve uma melhora na performance. Quando essa melhora de performance não é obtida, o engenheiro do sistema deve tomar uma dessas duas opções: (i) Modificar o código ou o conjunto

de treinamento e efetuar todos os testes novamente, ou (ii) Remodelar o *design* do agente conforme a figura 29 da seção 4.1.

Às vezes, os testes demonstram um ganho de performance no sistema, mas o novo agente inteligente começa a utilizar muitos recursos computacionais do sistema tais como CPU e memória. O sistema começa a apresentar lentidão no processamento, e em muitos casos pode atrasar processos que necessitam de agilidade. Recomenda-se a distribuição desse novo agente em uma outra CPU da rede. Após confirmar a melhora na *Medida de Performance Sistêmica*, uma nova versão é atribuída ao sistema. A remoção do código de outro agente selecionado na seção 4.1.2 é feita para que se possa repetir o processo de testes e avaliação.

4.2. Agent Learning Pattern – Um Design para Incluir Aprendizado em Agentes

Os conceitos de aprendizado devem ser modelados já na fase de *design* de um agente de software. Em sistemas complexos e abertos, agentes precisam de aprendizado para tomar decisões e se adaptar a mudanças para poder atingir os seus objetivos. Esta seção apresenta um *design* orientado a objetos utilizando a linguagem padrão de *design patterns* para guiar a inclusão de algoritmos de *machine learning*.

4.2.1.Intenção

O principal objetivo do *Agent Learning Pattern* é incluir algoritmos de *machine learning* em agentes com um *design* orientado a objetos. O *design* separa conceitos importantes de aprendizado em agentes, tais como: representação do conhecimento, algoritmo de *machine learning*, avaliador de performance, e gerador de exemplos para o aprendizado.

4.2.2.Contexto

Em ambientes complexos e abertos como a Internet, o sistema baseado em agentes deve ser capaz de se adaptar a situações desconhecidas e atingir o objetivo sistêmico. Técnicas de aprendizado são cruciais para sistemas multi-agente em

tais ambientes, pois são algoritmos que fornecem estratégias conhecidas para a implementação de agentes adaptáveis.

Mitchell (Mitchell, 1997) define *machine learning* como: "Diz-se que um programa de computador aprende a partir de uma experiência *E* em relação a uma classe de tarefas *T* e medida de performance *P*, se a sua performance nas tarefas *T*, medido por *P*, melhora com a experiência *E*". Conseqüentemente, esses são os principais aspectos de *design* de aprendizado em agentes: (i) uma representação do conhecimento adquirido pelo processo de aprendizado; (ii) o algoritmo de aprendizado; (iii) um gerador de exemplos para criar a experiência para o aprendizado; e (iv) o avaliador de performance para medir o processo de aprendizado.

4.2.3.Problema

O *design* de *machine learning* em arquiteturas orientada a objetos não é direto quando se precisa de um código reutilizável e de fácil manutenção. Como projetar um agente que utiliza vários algoritmos de *machine learning*? Como projetar diferentes geradores de exemplos para a técnica de aprendizado? Como projetar um monitor para avaliar a performance do processo de aprendizado?

4.2.4.Forças

- O *design* deve poder modelar qualquer técnica de *machine learning*
- Os aspectos de *design* de aprendizado devem ser mapeados com simplicidade para o *design* orientado a objetos
- O *design* deve melhorar o reuso e manutenção do código
- Algumas técnicas de aprendizado podem precisar em seu *design* de classes complementares
- Agentes de software devem ser capazes de utilizar o conhecimento adquirido para se adaptar as constantes mudanças em ambientes complexos e abertos.

4.2.5.Solução

Agentes de Software implementados em *frameworks* orientados a objetos (Telecom Italia-Jade, 2003; Sardinha et al., 2003a) normalmente utilizam herança

para implementar a abstração de agentes. A classe concreta *Agent* é um *design* OO típico de agente, e deve herdar de uma superclasse do *framework* orientado a objetos. Essa classe concreta deve implementar as funções básicas de um agente, tais como: coletar informação do ambiente, tratar eventos, processar mensagens, etc. A classe chamada *KnowledgeRepresentation* implementa a estrutura de dados do conhecimento do agente. A Figura 33 ilustra um diagrama de classes de um agente típico.

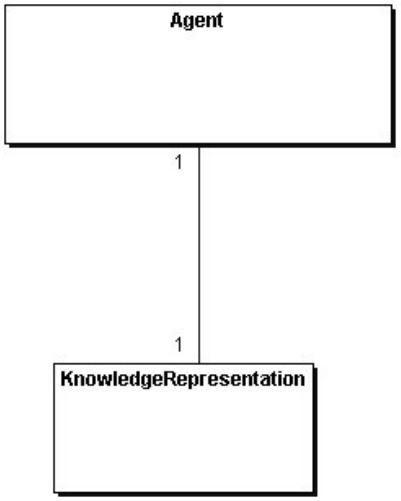

Figura 33: Um diagrama de Classes de um típico Agente.

A figura 34 mostra como incluir um algoritmo de *machine learning*, um monitor de performance, e um gerador de exemplos. O monitor de performance é codificado em uma classe chamada *PerformanceMeasure*, e é utilizado no processo de aprendizado para garantir que o agente está atingindo o seu objetivo de treinamento. O algoritmo de aprendizado é implementado na classe *LearningAlgorithm*. Essa classe é responsável por modificar a classe *KnowledgeRepresentation* após o processo de aprendizado. O gerador de exemplos para o algoritmo de aprendizado é codificado na classe *TrainingExperience*.

4.2.6.Estrutura

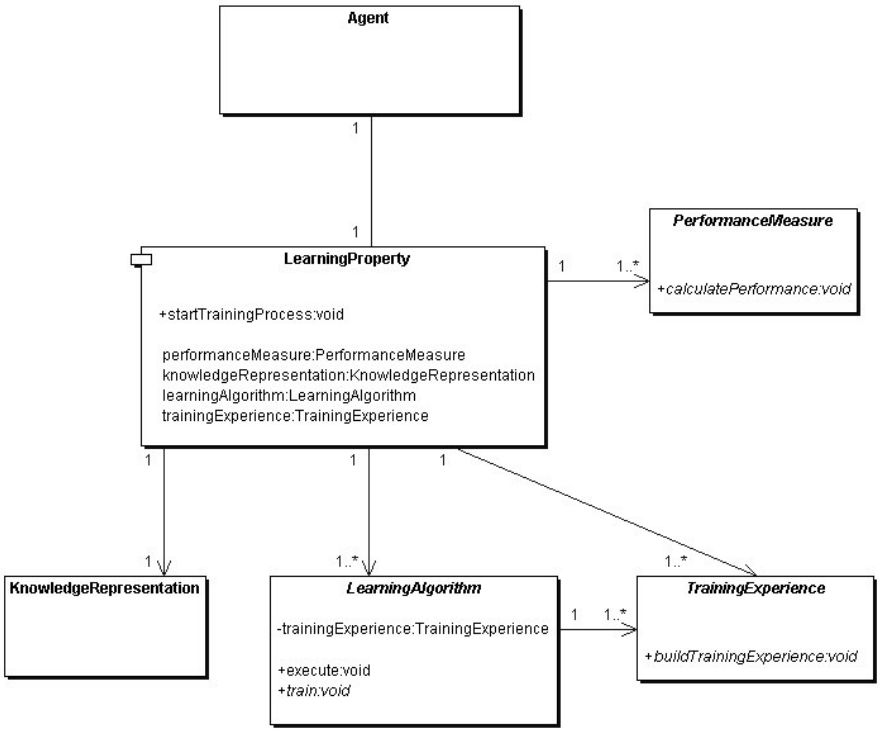

Figura 34: O diagrama de Classes do *Agent Learning Pattern*.

O *design pattern* possui quarto participantes principais e dois participantes clientes:

Participantes Principais:

LearningProperty

Define a classe que guarda as referencias para todas as outras classes relacionadas com o aprendizado.

PerformanceMeasure

O algoritmo que implementa regras para avaliar a performance do aprendizado. Um gerador de experimentos pode ser implementado nesta classe.

LearningAlgorithm

O algoritmo de *machine learning*.

TrainingExperience

O algoritmo que implementa o gerador de exemplos para o algoritmo de aprendizado

Participantes Clientes:

Agent

Define a classe que implementa os serviços básicos de um agente, tais como: coletar informação do ambiente, tratar de eventos, processar mensagens, etc.

KnowledgeRepresentation

A representação do conhecimento do agente.

4.2.7. Exemplo

O agente Preditor do exemplo da seção 4.1.2 utiliza o *Agent Learning Pattern* para incluir a técnica de predição de preços. As técnicas Media Móvel (Bowerman et al, 1993), Exponencial Suavizada (Bowerman et al, 1993), e PLS (Milidiu et al., 2005) foram implementadas como subclasses de *LearningAlgorithm*. A vantagem desse design é poder reutilizar todos as outras classes mesmo mudando o algoritmo de aprendizado. O teste de vários algoritmos de aprendizado é um processo comum para poder escolher a técnica com a melhor performance.

4.2.8. Dinâmica

A figura 35 apresenta a estrutura básica da dinâmica do *Agent Learning Pattern*. Vários eventos podem disparar o processo de aprendizado (Mitchell, 1997), incluindo a execução interna de uma ação, um levantamento de uma exceção, a troca de mensagens, ou através de informações coletadas no ambiente. A classe concreta *Agent* deve acessar o *LearningProperty* sempre que o agente quiser disparar o processo de aprendizado.

Por exemplo, o agente Preditor tem o principal objetivo de prever preços futuros baseado na seqüência de preços passadas. O método *buildTrainingExperience()* implementa o código que gera essa seqüência de preços. Estes preços são então utilizados como exemplos de treinamento para o algoritmo de aprendizado. O algoritmo de *machine learning* é codificado no método *train()*, mas é chamado pelo método *execute()*. Esse mesmo método também executa o método *calculatePerformance()* que é responsável por calcular

o erro de predição após o processo de aprendizado. Se o erro da predição for aceitável, o método *adaptKnowledge()* altera os atributos do conhecimento do agente.

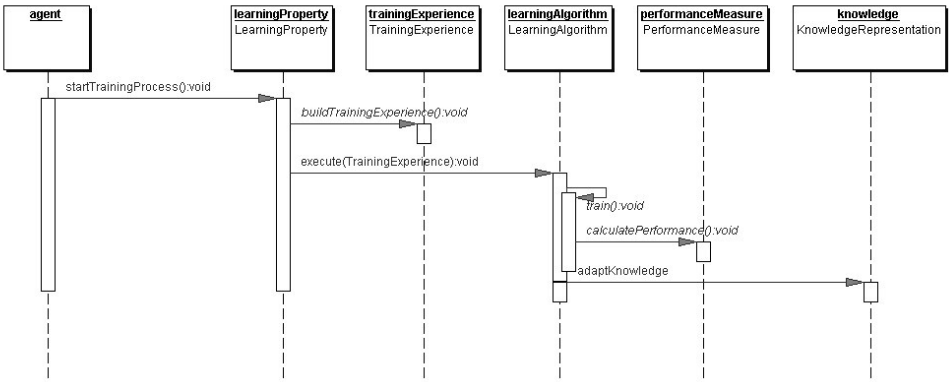

Figura 35: O diagrama de Seqüência do *Agent Learning Pattern*.

4.2.9.Implementação

A figura 36 apresenta o diagrama de classes do Preditor utilizando o ASYNC e o *Agent Learning Pattern*. As classes *PricePredictorAgent* e *PricePredictorAgentIP* são classes especializada do *framework* ASYNC, e implementam os serviços básicos do agente.

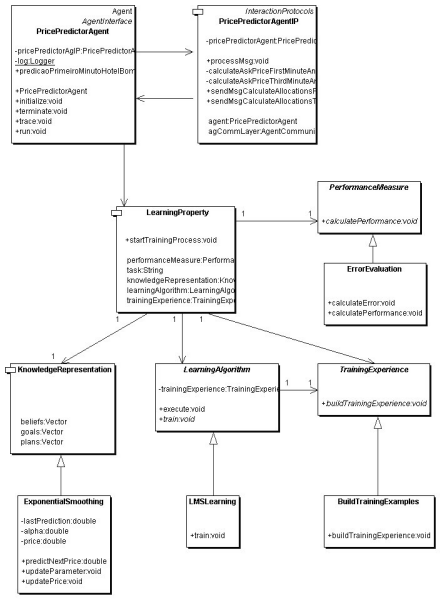

Figura 36: O diagrama de Classes do agente Preditor.

A classe *ExponentialSmoothing* implementa o conhecimento do agente, e por isso tem o *KnowledgeRepresentation* como superclasse. O algoritmo de treinamento é codificado na classe *LMSLearning* e a sua superclasse é o *LearningAlgorithm*. O conjunto de treinamento para o algoritmo *LMS* (Mitchell, 1997) é gerado pela classe *BuildTrainingExamples*. A classe *TrainingExperience* é superclasse de *BuildTrainingExamples*. A classe *ErrorEvaluation*, que herda de *PerformanceMeasure*, implementa a avaliação de performance do algoritmo de aprendizado.

Abaixo apresentamos alguns trechos de código Java para facilitar o entendimento do *design pattern*. A classe *LearningProperty* é referenciado pela classe *PricePredictorAgent* para acessar todos os componentes de aprendizado neste *design*. O método *startTrainingProcess()* (linha 9) executa o processo de geração do conjunto de treinamento (linha 10) e o algoritmo de *machine learning* (linha 11). A classe armazena em seus atributos as referencias para todas as outras classes relacionadas com o processo de aprendizado (linhas 3 até 6).

```
1. public class LearningProperty {
2.    private String task;
3.    private TrainingExperience trainingExperience;
4.    private LearningAlgorithm learningAlgorithm;
5.    private KnowledgeRepresentation knowledgeRepresentation;
6.    private PerformanceMeasure performanceMeasure;
7.    // Getters and Setters
8.     …
9.    public void startTrainingProcess(){
10.         trainingExperience.buildTrainingExperience();
11.         learningAlgorithm.execute(trainingExperience);
12.   }
13. }
```

O conhecimento desse agente foi modelado com uma função que implementa a técnica Exponencial Suavizada (Bowerman et al., 1993). A classe *ExponentialSmoothing* é uma especialização de *KnowledgeRepresentation*, e implementa a função abaixo no método *predictNextPrice()* (linha 5):

$$PredictedAskPrice(n) = \alpha * AskPrice(n-1) + (1 - \alpha) * PredictedAskPrice(n-1)$$

onde α é um numero entre 0 e 1,

e, n é o n-ésimo jogo.

```
1. public class ExponentialSmoothing extends KnowledgeRepresentation {
2.    private double lastPrediction;
3.    private double alpha;
4.    private double price;
5.    public double predictNextPrice(){
6.      double nextPrice = price*alpha + (1-alpha)*lastPrediction;
7.      lastPrediction = nextPrice;
8.      return(nextPrice);
9.    }
10.   public void updateParameter(double alpha){
11.     this.alpha=alpha;
12.   }
13.   public void updatePrice(double price){
14.     this.price=price;
16.   }
17. }
```

A classe *BuildTrainingExamples* é uma especialização de *TrainingExperience*, e implementa o processo de geração do conjunto de treinamento. O método *buildTrainingExperience()* (linha 3) implementa uma consulta no banco de dado com os preços de partidas antigas. Essa seqüência de preços é armazenada no vetor *askPriceHstory* (linha 2).

```
1. public class BuildTrainingExamples extends TrainingExperience {
2.    public Vector askPriceHistory;
3.    public void buildTrainingExperience() {
4.      // JDBC code
5.       …
6.    }
7. }
```

O agente Preditor utiliza o algoritmo *Least Mean Squares* (LMS) (Mitchell, 1997) para adaptar o valor de alfa (α) na classe *ExponentialSmoothing*. A classe que implementa esse algoritmo é o *LMSLearningAlgorithm*, e possui o *LearningAlgorithm* como superclasse. O método *train()* (linha 11) codifica o algoritmo LMS.

```
1. public class LMSLearning extends LearningAlgorithm {
2.    private BuildTrainingExamples bte;
3.    private ExponentialSmoothing es;
4.    private double alpha;
5.    private double lastAlpha;
6.    private double beta;
7.    private double predictedPrice;
8.    private double lastAskPrice;
9.    es.updateParameter(20);
10.   alpha=20;
11.   public void train() {
12.     for(int i=0;i<bte.askPriceHistory.size();i++){
13.       lastAskPrice=
14.         ((Double)bte.askPriceHistory.elementAt(i)).doubleValue();
```

```
15.       es.updatePrice(lastAskPrice);
16.       predictedPrice = es.predictNextPrice();
17.       lastAlpha=alpha;
18.       alpha=lastAplha+(beta*(lastAskPrice-predictedPrice));
19.    }
20. }
```

A classe *ErrorEvaluation* é uma especialização da classe *PerformanceMeasure*, e a classe *LeaningProperty* utiliza esta classe para calcular o erro entre o preço previsto e o preço realizado. O método *calculateError()* (linha 5) implementa esse cálculo.

```
1. public class ErrorEvaluation extends PerformanceMeasure {
2.    private double predictedPrice;
3.    private double askPrice;
4.    private double error;
5.    public void calculateError() {
6.          error = Math.abs(predictedPrice-askPrice)/askPrice;
7.    }
8.    …
9. }
```

4.2.10.Conseqüências

Generalidade e Uniformidade. O *Learning design pattern* fornece uma solução uniforme e geral para qualquer técnica de *machine learning*.

Reuso. O padrão apresenta um *design* modular para incluir aprendizado em agentes de software que pode ser reutilizado e refinado para diferentes contextos e aplicações.

Uma melhor Separação de Concerns. O padrão é totalmente separado de outros *concerns* de agentes, tais como interação e autonomia.

Implementação Direta. O padrão implementa abstrações de aprendizado diretamente para classes orientados a objetos.

4.2.11.Usos Conhecidos

Um *design* para incluir algoritmos de *machine learning* é apresentado no capítulo 1 (seção 1.2.5) do livro Machine Learning (Mitchell, 1997). Quatro módulos são apresentados nesse *design*: (i) *Sistema de Performance* – Módulo para resolver uma dada tarefa de performance. Esse módulo recebe a instancia de

um novo problema como entrada e produz um *trace* da solução; (ii) *O Crítico* – Esse módulo recebe como entrada um histórico ou *trace* da solução e gera exemplos de treinamento; (iii) *O Generalizador* – Esse módulo recebe como entrada exemplos de treinamento e produz uma hipótese; (iv) *Gerador de Experimentos* – Esse módulo recebe como entrada uma hipótese corrente e gera novos problemas para o *Sistema de Performance* explorar. No *design* do ASYNC *Agent Learning Pattern*, o módulo *O Crítico* é codificado na classe *TrainingExperience*. O módulo *O Generalizador* é codificado na classe *LearningAlgorithm*, e este módulo é responsável por armazenar a hipótese corrente na classe *KnowledgeRepresentation*. O módulo *Gerador de Experimentos* e *Sistema de Performance* são codificados na classe *PerformanceMeasure*.

O *Agent Learning Pattern* foi utilizado em cinco implementações: (i) A aplicação *Bundles.com* (Sardinha, 2001); (b) A aplicação *Learn Agent Player* (Sardinha et al., 2003b); (c) A aplicação *LearnAgents* (Sardinha et al., 2004c; Sardinha et al., 2005a); (d) Um sistema multi-agente para gerenciar submissões de artigos e o processo de seleção em *workshops* e conferências (Garcia, 2004a); e, A aplicação *LearnAgentsSCM* do capítulo 4.

4.2.12.Padrões Relacionados

Learning Aspect. (Garcia et al., 2004b) Uma mesma versão do *Learning Pattern* pode ser implementada como um aspecto (Kiczales et al., 1997) e melhorar a separação de *concerns* (Garcia, 2004). Um aspecto pode substituir a classe *LearningProperty*, conectar pontos de execução (eventos) em diferentes classes do agente, e identificar quando o processo de aprendizado deve ser acionado. Algumas vantagens adicionais podem ser encontradas: (i) Transparência – O uso de aspectos torna o código mais elegante e poderoso para incluir aprendizado em agentes de software de uma forma mais transparente (Garcia et al., 2004b); A descrição de que classes precisam ser alteradas pelo aprendizado está presente no aspecto e as classes monitoradas não precisam ser modificadas; (ii) Facilidade de evolução – Conforme o sistema multi-agente evolui, novas classes de agentes precisam ser monitoradas e alteradas pelo processo de

aprendizado; Os programadores do sistema precisam apenas adicionar novos *pointcuts* ao aspecto para implementar novas funcionalidades.