Referências

1. ABLE WEBSITE. **ABLE: Agent Building and Learning Environment**. URL: http://www.alphaworks.ibm.com/tech/able
2. AGHA, G.; HEWITT, C. Concurrent Programming Using Actors. In A. Yonezawa and M. Tokoro (eds.), **Object-Oriented Concurrent Programming**, MIT Press, 1988, pages 37-53.
3. AKSIT, M. et al. Abstracting Object Interactions Using Composition Filters. **Proceedings of the Workshop on Object-Based Distributed Programming at ECOOP'93**, Springer-Verlag, 1993, pp. 152–184.
4. ALEXANDER, C. **The Timeless Way of Building**. Oxford University Press, New York, 1979.
5. ALEXANDER, C. et al. **A Pattern Language**. Oxford University Press, New York, 1977.
6. ALWIS, B. et al. Coding Issues in AspectJ. **Proceedings of the International Workshop on Advanced Separation of Concerns at OOPSLA'00**, November 2000.
7. AMANDI, A. **Programação de Agentes Orientada a Objetos**. Tese de Doutorado, Instituto de Informática, UFRGS, Porto Alegre, RS, 1997. (In Portuguese)
8. AMANDI, A.; PRICE, A. Building Object-Agents from a Software Meta-Architecture. In: **Advances in Artificial Intelligence**, Lecture Notes in Artificial Intelligence, vol. 1515, Springer-Verlag, 1998.
9. AOSD WEBSITE. http://aosd.net/
10. ARIDOR, Y.; LANGE, D. Agent Design Patterns: Elements of Agent Application Design. **Proceedings of the 2nd International Conference on Autonomous Agents (Agents '98)**, ACM Press, 1998, pp. 108-115.
11. ASPECT-ORIENTED SOFTWARE ENGINEERING SPECIAL INTEREST GROUP. Computing Department, Lancaster University. http://www.comp.lancs.ac.uk/computing/aop/
12. ASPECTJ TEAM. The AspectJ Programming Guide. March, 2003. http://eclipse.org/aspectj/
13. ASPECTWERKZ WEBSITE. Simple, Dynamic, Lightweight and Powerful AOP for Java. http://aspectwerkz.codehaus.org/

14. BANDI, B. K.; VAISHNAVI, V. K.; TURK, D. E. Predicting Maintenance Performance Using Object-Oriented Design Complexity Metrics. **IEEE Transactions on Software Engineering**, 29 (1), January, 2003, pp 77-87.

15. BASILI, V.; BRIAND, L.; MELO, W. A Validation of Object-Oriented Design Metrics as Quality Indicators. **IEEE Transactions on Software Engineering**, 22 (10), October 1996, pp. 751-761.

16. BASILI, V.; CALDIERA, G.; ROMBACH, H. The Goal Question Metric Approach. **Encyclopedia of Software Engineering**, September 1994, 2(9): 528-532. John Wiley & Sons, 1994

17. BASILI, V.; SELBY, R.; HUTCHEN, D. Experimentation in Software Engineering. **IEEE Transactions on Software Engineering**, 1986, 12(7):733-743

18. BASS, L.; CLEMENTS, P.; KAZMAN, R. **Software Architecture in Practice**. Addison-Wesley, 1997.

19. BAUMER, D.; RIEHLE, D.; SIBERSKI, W.; WOLF, M. Role Object. **Proceedings of the 4th Annual Conference on the Pattern Languages of Programs**, Monticello, Illinois, USA, September 2-5, 1997.

20. BELLIFEMINE, F.; POGGI, A.; RIMASSI, G. JADE: A FIPA-Compliant Agent Framework. **Proceedings of the Practical Applications of Intelligent Agents and Multi-Agents**, April 1999; pp. 97-108.

21. BERGMANS, L.; AKSIT, M. Composing Crosscutting Concerns Using Composition Filters. **Communications of the ACM**, 44(10):51–57, October 2001.

22. BIGUS, J.; BIGUS, J. **Constructing Intelligent Agents Using Java: Professional Developer's Guide Series**. 2nd Edition, John Wiley & Sons, 2001.

23. BOEHM, B. **Software Engineering Economics**. Prentice-Hall, 1981.

24. OMG. **Unified Modeling Language Specification**. Object Management Group, Version 1.4, 2001.

25. BORLAND TOGETHERSOFT WEBSITE. URL: http://www.togethersoft.com

26. BRACHA, G.; COOK, W. Mixin-based Inheritance. **Proceedings of ACM Symposium on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA'90)**, ACM Press, October 1990. SIGPLAN Notices, volume 25, number 10, pp. 303-311.

27. BRADSHAW, J. An Introduction to Software Agents. **Software Agents**, J. Bradshaw (ed.). AAAI /MIT Press, 1997.

28. BRADSHAW, J.; DUTFIELD, S.; BENOIT, P.; WOOLLEY, J. KaoS: Toward an Industrial-Strength Generic Agent Architecture. **Software Agents**, J. M. Bradshaw (ed.). AAAI/MIT Press: Cambridge, MA, 1996.

29. BRIAND, L.; DALY, J.; WÜST, J. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. **Empirical Software Engineering Journal**, 3 (1), 1998, pp 65-117.

30. BRIAND, L.; DALY, J.; WÜST, J. A Unified Framework for Coupling Measurement in Object-Oriented Systems. **IEEE Transactions on Software Engineering**, 25 (1), 1999, pp 91-121.

31. BRIAND, L.; EL EMAM, K.; MORASCA, S. **Theoretical and Empirical Validation of Software Product Measures**. Technical Report ISERN-95-03, Fraunhofer Institute for Experimental Software Engineering, Germany, 1995.

32. BRIAND, L.; MORASCA, S.; BASILI, V. Property-Based Software Engineering Measurement. **IEEE Transactions of Software Engineering**, 22 (1), 1996, pp. 68-86.

33. BRIOT, J.; GASSER, L. Agents and Concurrent Objects. **IEEE Concurrency**, Special Issue on Actors and Agents, 1998.

34. BRUGALI, D.; SYCARA, K. A Model for Reusable Agent Systems. In: **Implementing Application Frameworks – Object-Oriented Frameworks at Work**, M. Fayad et al. (editors), John Wiley & Sons, 1999.

35. BUSCHMANN, F. et al. **Pattern-Oriented Software Architecture: A System of Patterns**. John Wiley & Sons, 1996.

36. CABRI, G.; LEONARDI, L.; ZAMBORNELLI, F. MARS: A Programmable Coordination Architecture for Mobile Agents. **IEEE Internet Computing**, Vol. 4, No. 4, pp. 26-35, July-August 2000.

37. CAMACHO, D. Coordination of Planning Agents to Solve Problems in the Web. **AI Communications**, IOS Press, Vol. 16 (4), November, 2003, pp. 309-311.

38. CAMACHO, D.; BORRAJO, D.; MOLINA, J. Intelligent Travel Planning: A MultiAgent Planning System to Solve Web Problems in the e-Tourism Domain. **International Journal on Autonomous Agents and Multiagent Systems**, Kluwer Academic Publishers, Vol. 4, n. 4, 2001, pp. 387-392.

39. CAMACHO, D.; BORRAJO, D.; MOLINA, J.; ALLER, R.. MAPWEB: Cooperation between Planning Agents and Web Agents. **Information & Security: An International Journal**, Special Issue on Agent-based Technologies, Volume 8, Number 2, pages 209-238, 2002.

40. CHAUHAN, D. Developing Coherent Multiagent Systems Using Jafmas. **Proc. International Conference on Multi Agent Systems, ICMAS98**, Cite des Sciences – La Villette, Paris, France, July 1998.

41. CHAVEZ, C. **A Model-Driven Approach to Aspect-Oriented Design**. PhD Thesis, Computer Science Department, PUC-Rio, April 2004, Rio de Janeiro, Brazil.

42. CHAVEZ, C.; GARCIA, A.; LUCENA, C. Some Insights on the Use of AspectJ and Hyper/J. **Proceedings of the Tutorial and Workshop on Aspect-**

Oriented Programming and Separation of Concerns, Lancaster, UK, August 2001

43. CHAVEZ, C.; LUCENA, C. Design-level Support for Aspect-oriented Software Development. **Proceedings of the Workshop on Advanced Separation of Concerns in Object-Oriented Systems at OOPSLA'2001**, Tampa Bay, Florida, USA, October 14, 2001.

44. CHAVEZ, C.; LUCENA, C. A Theory of Aspects for Aspect-Oriented Software Development. **Proceedings of the 17th Brazilian Symposium on Software Engineering**, Manaus, Brazil, October 2003.

45. CHEN, P. The Entity Relationship Model – Towards a Unified View of Data. **ACM Transactions on Database Systems**, vol. 1, no1. March 1976, 9-36.

46. CHIDAMBER, S.; KEMERER, C. A Metrics Suite for Object Oriented Design. **IEEE Transactions on Software Engineering**, 20 (6), June 1994, pp. 476-493.

47. CLARKE, S.; WALKER, R. Composition Patterns: An Approach to Designing Reusable Aspects. **Proceedings of the 23rd International Conference on Software Engineering** (Toronto, Canada; 12--19 May), pp. 5--14, 2001.

48. CLARKE, S.; WALKER, R. Towards a Standard Design Language for AOSD. **Proceedings of the 1st International Conference on Aspect-Oriented Software Development**, Enschede,The Netherlands, April 2002,pp. 113-119.

49. CLARKE, S.; WALKER, R. Mapping Composition Patterns to AspectJ and Hyper/J. **Proceedings of the International Conference on Software Engineering (ICSE'01)**, May 2001.

50. COPLIEN, J.; SCHMIDT, D. **Pattern Languages of Program Design**. Addison-Wesley, (Software Patterns Series), 1995.

51. COSTA, A. **An Aspect-Oriented Software Architecture for Traffic Simulators**. Master's Dissertation, University of Sao Paulo, October 2003. (In Portuguese)

52. CRISTIAN, F. Exception Handling and Software Fault Tolerance. **IEEE Transactions on Computers**, C-31, 1982, pp. 531-540.

53. CRISTIAN, F. Exception Handling. In **Dependability of Resilient Computers**, T. Anderson (ed.), Blackwell Scientific Publications, 1989, p. 68-97.

54. CRISTIAN, F. Exception Handling and Tolerance of Software Faults. In **Software Fault Tolerance**, M. Lyu (ed.), John Wiley & Sons, 1995, p. 81-107.

55. CZARNECKI, K.; EISENECKER, U. **Generative Programming: Methods, Tools, and Applications**. Addison-Wesley, 2000.

56. DEBENHAM, J.; HENDERSON-SELLERS, B.; JENNINGS, N.; ODELL, J. (eds). **Proceedings of the OOPSLA 2002 Workshop on Agent-Oriented Methodologies**. COTAR, 2002.

57. DELOACH, S.; WOOD, M.; SPARKMAN, C. Multiagent Systems Engineering. **International Journal of Software Engineering and Knowledge Engineering**, 11(3):231--258, 2001.

58. DEMAZEAU, Y.; MULLER, J. From Reactive to Intentional Agents. **Decentralized Artificial Intelligence** 2. pp.3-14. 1991.

59. DEUGO, D.; WEISS, M.; KENDALL, E. Reusable Patterns for Agent Coordination. In: Omicini, A., Zambonelli, F., Klusch, M., and Tolksdorf, R. (eds.), **Coordination of Internet Agents: Models, Technologies, and Applications**, Springer, 2001.

60. DIJKSTRA, E. **A Discipline of Programming**. Prentice Hall, Englewood Cliffs, NJ, 1976.

61. D'INVERNO, M.; LUCK, M. **Understanding Agent Systems**. Springer, 2001.

62. DRIVER, C. **Evaluation of Aspect-Oriented Software Development for Distributed Systems**. Masters Thesis, University of Doublin, September 2002.

63. EARLY ASPECTS RESEARCH GROUP. Departamento de Informática, Universidade Nova de Lisboa. http://ctp.di.fct.unl.pt/early-aspects/

64. EARLY ASPECTS WEBSITE. http://early-aspects.net/

65. EKDAHL, B. How Autonomous is an Autonomous Agent? **Proceedings of the 5th Conference on Systemic, Cybernetics and Informatics (SCI 2001)**, July 22-25, 2001, Orlando, Florida, USA.

66. ELAMMARI, M.; LALONDE, W. An Agent-Oriented Methodology: High-level and Intermediate Models. In: Wagner, G., Yu, E. (eds).: **Proceedings of the 1st International Workshop on Agent-Oriented Information Systems**, 1999.

67. ELRAD, T.; FILMAN, R.; BADER, A. Aspect-Oriented Programming. **Communications of the ACM**, 44(10):29–32, October 2001.

68. FENTON, N.; PFLEEGER, S. **Software Metrics: A Rigorous and Practical Approach**. 2.ed. London: PWS, 1997.

69. FERBER, J. **Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence**. Addison Wesley Longman, 1999.

70. FERBER, J.; GUTKNECHT, O. A meta-model for the analysis and design of organizations in multi-agent systems. **Proceedings of the 3rd International Conference on Multi-Agent Systems (ICMAS'98)**, IEEE Computer Society, 1998, pp. 128-135

71. FILMAN, R. What Is Aspect-Oriented Programming, Revisited. **Proceedings of the Workshop on Advanced Separation of Concerns at ECOOP'01**, June 2001.

72. FININ, T. et al. KQML as an Agent Communication Language. **Proceedings of the Third International Conference on Information and Knowledge Management**, 1994, pp. 456-463.

73. FINKELSTEIN, A. et al. Viewpoints: a Framework for Integrating Multiple Perspectives in System Development. **International Journal of Software Engineering and Knowledge Engineering**, 2(1):31-37; 1992.

74. FIPA. Foundation for Intelligent Physical Agents. http://www.fipa.org.

75. FIPA. Foundation for Intelligent Physical Agents, Agent Communication Technical Committee. **Agent Communication Language - FIPA'99 Draft Specification**, 1999. http://www.fipa.org.

76. FLEURY, M; REVERBEL, F. The JBoss Extensible Server. **Proceedings of the International Middleware Conference 2003**, vol. 2672 of LNCS, pp. 344-373, Springer-Verlag, Rio de Janeiro, Brazil, June 2003.

77. FOWLER, M. Dealing with Roles. **Proceedings of the 4th Annual Conference on the Pattern Languages of Programs**, Monticello, Illinois, USA, September 2-5, 1997.

78. FUGGETTA, A.; PICCO, G.; VIGNA, G. Understanding Code Mobility. **IEEE Transactions on Software Engineering**, vol.24, No.5, pp.342-361, 1998.

79. GAMMA, E. et al. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, Reading, MA, 1995.

80. GARCIA, A. et al. An Aspect-Based Approach for Developing Multi-Agent Object-Oriented Systems. **Proceedings of the 15th Brazilian Symposium on Software Engineering**, Rio de Janeiro, Brazil, October 2001, pp. 177-192. (Nomination for the Best Paper Award)

81. GARCIA, A. et al. Promoting Advanced Separation of Concerns in Intra-Agent and Inter-Agent Software Engineering. **Proceedings of the Advanced Separation of Concerns Workshop at OOPSLA'2001**, Tampa, USA, October 2001.

82. GARCIA, A. et al. Engineering Multi-Agent Systems with Aspects and Patterns. **Journal of the Brazilian Computer Society**, Special Issue on Software Engineering and Databases, N. 1, Vol. 8, July 2002, pp. 57-72.

83. GARCIA, A. et al. **Agents and Objects: An Empirical Study on Software Engineering**. Technical Report 06-03, Computer Science Department, PUC-Rio, February 2003. Available at ftp://ftp.inf.puc-rio.br/pub/docs/techreports/

84. GARCIA, A. et al. Agents and Objects: An Empirical Study on the Design and Implementation of Multi-Agent Systems. **Proceedings of the 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'03) at ICSE 2003**, Portland, USA, May 2003, pp. 11-20.

85. GARCIA, A.; BEDER, D.; RUBIRA, C. An Exception Handling Mechanism for Developing Dependable Object-Oriented Software Based on a Meta-Level Approach. **Proceedings of the 10th IEEE International Symposium on Software Reliability Engineering (ISSRE'99)**, Florida, USA, November 1999.

86. GARCIA, A.; BEDER, D.; RUBIRA, C. An Exception Handling Software Architecture for Developing Fault-Tolerant Software. **Proceedings of the 5th IEEE High Assurance Systems Engineering Symposium (HASE 2000)**, Albuquerque, New Mexico, USA, November 2000.

87. GARCIA, A., BEDER, D., RUBIRA, C. A Unified Meta-Level Software Architecture for Concurrent and Sequential Exception Handling. **The Computer Journal**, Special Issue on High Assurance Systems Engineering, Vol. 44, No. 6, January 2002, pp. 569-587.

88. GARCIA, A; CHAVEZ, C; LUCENA, C. Aspect-Oriented Software Development. Tutorial Notes, **17th Brazilian Symposium on Software Engineering**, Manaus, Brazil, October 2003. (In Portuguese)

89. GARCIA, A.; CORTÉS, M.; LUCENA, C. A Web Environment for the Development and Maintenance of E-Commerce Portals based on a Groupware Approach. **Proceedings of the Information Resources Management Association International Conference (IRMA'01)**, Toronto, May 2001, pp. 722-724.

90. GARCIA, A.; LUCENA, C. An Aspect-Based Object-Oriented Model for Multi-Agent Systems. **Proceedings of the 2nd Advanced Separation of Concerns Workshop at ICSE'2001**, Toronto, Canada, May 2001.

91. GARCIA, A.; LUCENA, C. Software Engineering for Large-Scale Multi-Agent Systems – SELMAS 2002. **ACM Software Engineering Notes**, September 2002; Vol. 27, N. 5, pp. 82-88.

92. GARCIA, A.; LUCENA, C. Agents and Objects in Software Engineering: An Aspect-Oriented Approach. **Proceedings of the Doctoral Symposium at ECOOP 2003**, Darmsdat, Germany, July 2003.

93. GARCIA, A.; LUCENA, C.; CASTRO, J.; OMICINI, A.; ZAMBONELLI, F (editors). **Software Engineering for Large-Scale Multi-Agent Systems**. Lecture Notes in Computer Science, vol. 2603, Springer-Verlag, April 2003.

94. GARCIA, A.; LUCENA, C.; COWAN, D. **Agents in Object-Oriented Software Engineering**. Technical Report CS-2001-07, Computer Science Department, University of Waterloo, Waterloo, Canada, March 2001.

95. GARCIA, A.; LUCENA, C.; COWAN, D. Agents in Object-Oriented Software Engineering. **Software: Practice and Experience**, Elsevier, May 2004, pp. 1-32. (To Appear)

96. GARCIA, A.; RUBIRA, C. An Architectural-based Reflective Approach to Incorporating Exception Handling into Dependable Software. In: A. Romanovsky, C. Dony, J. L. Knudsen, A. Tripathi (Eds), **Advances in Exception Handling Techniques**, Springer-Verlag, LNCS-2022, April 2001.

97. GARCIA, A.; RUBIRA, C.; ROMANOVSKY, A.; XU, J. A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. **Journal of Systems and Software**, Elsevier, Vol. 59, No. 6, November 2001, pp. 197-222.

98. GARCIA, A.; SANT'ANNA, C.; CHAVEZ, C.; SILVA, V.; LUCENA, C.; STAA, A. Separation of Concerns in Multi-Agent Systems: An Empirical Study. In: C. Lucena et al (Eds)., **Software Engineering for Multi-Agent Systems II**, Springer-Verlag, LNCS 2940, March 2004, pp. 49-72.

99. GARCIA, A.; SARDINHA, J.; LUCENA, C.; LEITE, J.; MILIDIÚ, R.; CASTRO, J.; ROMANOVSKY, A.; GRISS, M.; LEMOS, R.; PERINI, A. Software Engineering for Large-Scale Multi-Agent Systems – SELMAS 2003. (Post-Workshop Report) **ACM Software Engineering Notes**, Vol. 28, Number 6, November 2003.

100. GASSER, L.; BRIOT, J. Object-Oriented Concurrent Programming and Distributed Artificial Intelligence. **Distributed Artificial Intelligence: Theory and Praxis**, Kluwer Academic Publishers, 1992.

101. GEDENRYD, H. **Universal Composition — An Optimal Scheme for Structuring (Software) Systems**. This paper is available on-line at: http://www.lucs.lu.se/People/Henrik.Gedenryd/Squeak/UC/UCpaper.pdf

102. GEHANI, N. Exceptional C or C with Exceptions. **Software: Practice and Experience**, 22, 827-848 (1992).

103. GELERNTER, D. Generative Communication in Linda. **ACM Transactions on Programming Languages and Systems**, vol. 7 - No.1, pp 80-112, 1985

104. GENESERETH, M.; FIKES, R. **Knowledge Interchange Format, Version 3.0, Reference Manual**. Technical Report Logic-92-1, Computer Science Department, Stanford University 1992. http://logic.stanford.edu/kif/kif.html

105. GENESERETH, N.; KETCHPEL, S. Software Agents. **Communications of the ACM**, v. 37, n. 7, July 1994.

106. GHEZZI, C. et al. **Fundamentals of Software Engineering**. Prentice-Hall International, London, 1991.

107. GOLM, M.; KLEINODER, J. MetaXa and the Future of Reflection. **Proceedings of the Workshop on Reflective Programming in C++ and Java**, 1998.

108. GOTTLOB, G.; SCHREFL, M.; AND ROCK, B. Extending Object-Oriented Systems with Roles. **ACM Transaction on Information Systems**, Vol. 14, No. 3, July, 1996, pp. 268 - 296.

109. GRECU, D., BROWN, D. Dimensions of Learning in Agent-based Design. **Proceedings of the Workshop on Machine Learning in Design**, 4th International Conference on AI in Design, Stanford, CA, June 1996.

110. GROSSO, W. Aspect-Oriented Programming and AspectJ. **Dr. Dobbs Journal**. August 2002. http://www.ddj.com/articles/2002/0208/

111. GRUNDY, J. Aspect-Oriented Requirements Engineering for Component-based Software Systems. **Proceedings of the 4th IEEE International Symposium on Requirements Engineering**, IEEE Computer Society Press, 1999, pp. 84-91.

112. GUESSOUM, Z.; BRIOT, J. From Active Objects to Autonomous Agents. **IEEE Concurrency**, Special Series on Actors and Agents, Vol. 7, N. 3, 1999, pp. 68-76.

113. GUTTMAN, R.; MOUKAS, A.; MAES, P. Agent-Mediated Electronic Commerce: A Survey. **Knowledge Engineering Review**, 13(2):147-159, 1998.

114. HANENBERG, S.; UNLAND, R.; SCHMIDMEIER, A. AspectJ Idioms for Aspect-Oriented Software Construction. **Proceedings of the 8th European Conference on Pattern Languages of Programming and Computing (EuroPlop'03)**, Irsee, Germany, June 2003.

115. HANNEMANN, J.; KICZALES, G. Design Pattern Implementation in Java and AspectJ. **Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)**, November 2002, pp. 161-173.

116. HARRISON, W.; OSSHER, H. Subject-Oriented Programming: A Critique of Pure Objects. **Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'93)**, 1993, pp. 411- 428.

117. HAYDEN, S.; CARRICK, C.; YANG, Q. Architectural Design Patterns for Multiagent Coordination. **Proceedings of the International Conference on Agent Systems '99 (Agents'99)**, Seattle, WA, May 1999.

118. HENDERSON-SELLERS, B. **Object-Oriented Metrics: Measures of Complexity**. Prentice Hall, 1996.

119. HUHNS, M.; SINGH, M. (Eds.). Agents and Multiagent Systems: Themes, Approaches, and Challenges. **Readings in Agents**, Chapter 1, Morgan Kaufmann Publishers, Inc., San Francisco, California, pp. 1-23.

120. IBM: AGLETS SOFTWARE DEVELOPMENT KIT HOME PAGE. http://www.trl.ibm.co.jp/aglets/index.html, 1999.

121. IBM RESEARCH: SUBJECT-ORIENTED PROGRAMMING GROUP. **Subject-oriented Programming and Design Patterns**. www.ibm.research/sop

122. IGLESIAS, C.; GARRIJO, M.; GONZALEZ, J. A Survey of Agent-Oriented Methodologies. **Proceedings of the 5th International Workshop on Intelligent Agents: Agent Theories, Architectures, and Languages (ATAL-98)**, Paris, France, July 1998; 317-330.

123. INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. IEEE **Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries**. New York, 1990.

124. JACOBSON, I. **Object-Oriented Software Engineering**. Addison-Wesley, 1992.

125. JACOBSON, I. Use Cases and Aspects – Working Seamlessly Together. **Journal of Object Technology**, vol. 2, no. 4, July-August 2003, pp. 7-28.

126. JARZABEK, S. Design of Flexible Static Program Analyzers with PQL. **IEEE Transactions of Software Engineering**, vol. 24, No. 3, March 1998.

127. JENNINGS, N. Agent-Oriented Software Engineering. **Proceedings of the 12th International Conference on Industrial and Engineering Applications of Artificial Intelligence**, 1999, pp. 4-10.

128. JENNINGS, N.; WOOLDRIDGE, M. Agent-Oriented Software Engineering. **Handbook of Agent Technology**, J. Bradshaw (ed.). AAAI/MIT Press, 2000.

129. JUAN, T.; STERLING, L. A Meta-Model for Intelligent Adaptive Multi-Agent Systems in Open Environments. **Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems - AAMAS 2003**, July 2003, Melbourne, Australia.

130. KANG, K. et al. **Feature-oriented domain analysis (FODA) feasibility study**. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

131. KARAGEORGOS, A.; THOMPSON, S.; MEHANDJIEV, N. Semi-Automatic Design of Agent Organisations. **Proceedings of the ACM Symposium on Applied Computing**, Madrid, Spain, March 2002, pp. 306-313.

132. KARNIK, N.; TRIPATHI, A. Design Issues in Mobile-Agent Programming Systems. **IEEE Concurrency**, vol. 6, n. 3, 1998, pp.52-61.

133. KARNIK, N.; TRIPATHI. Security in the Ajanta Mobile Agent System. **Software - Practice and Experience**, January 2001.

134. KAZMAN, R.; ABOWD, G.; BASS, L.; CLEMENTS, P. Scenario-Based Analysis of Software Architecture. **IEEE Software**, vol. 13, n. 6, November 1996, pp. 47-55.

135. KEITH, J. Towards a Distributed, Environment-Centered Agent Framework. In: N. Jennings and Y. Lesperance (eds.): **Intelligent Agents VI**, Proceedings of the Sixth International Workshop on Agent Theories, Architectures, and Languages (ATAL-99), Lecture Notes in Articial Intelligence, Springer-Verlag, Berlin, 2000.

136. KENDALL, E. Role Model Designs and Implementations with Aspect-oriented Programming. **Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)**, ACM Press, 1999, pp. 353-369.

137. KENDALL, E. et al. A Framework for Agent Systems. **Implementing Application Frameworks – Object-Oriented Frameworks at Work**, M. Fayad et al. (eds.). John Wiley & Sons: 1999.

138. KERSTEN, A.; MURPHY, G. Atlas: A Case Study in Building a Web-based Learning Environment Using Aspect-Oriented Programming. **Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)**, November 1999.

139. KICZALES, G. et al. Aspect-Oriented Programming. **Proceedings of the European Conference on Object-Oriented Programming - ECOOP'97**, LNCS (1241), Springer-Verlag, Finland., June 1997.

140. KICZALES, G. et al. An Overview of AspectJ. **Proceedings of the European Conference on Object-Oriented Programming (ECOOP'01)**, Budapest, Hungary, 2001.

141. KIENZLE, J.; GUERRAOUI, R. AOP: Does it Make Sense? The Case of Concurrency and Failures. **Proceedings of the European Conference on Object-Oriented Programming (ECOOP'02)**, July 2002.

142. KINNY, D.: The # Calculus: An Algebraic Agent Language. In: **Intelligent Agents VIII**, Springer-Verlag, Vol. 2333, (2002): 32-50.

143. KINNY, D.; GEORGEFF, M. Modelling and Design of Multi-Agent System. **Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages (ATAL'96)**, Budapest, Hungary, August 1996, pp. 1-20.

144. KITCHENHAM, B.; PFLEEGER, S.; FENTON, N. Towards a Framework for Software Measurement Validation. **IEEE Transactions on Software Engineering** 12, 929-944, December 1995.

145. KOLB, M. A Cooperation Language. **Proceedings of the First International Conference on Multi-Agent Systems (ICMAS'95)**, June 1995, pp. 233-238.

146. KRISTENSEN, B.; OSTERBYE, K. Roles: Conceptual Abstraction Theory and Practical Language Issues. **Theory and Practice of Object Systems (TAPOS)**, Special Issue on Subjectivity in Object-Oriented Systems, 1996.

147. KULESZA, U.; GARCIA, A.; LUCENA, C. Generating Aspect-Oriented Agent Architectures. **Proceedings of the 3rd Workshop on Early Aspects - Aspect-Oriented Requirements Engineering and Architecture Design,** 3rd International Conference on Aspect-Oriented Software Development**,** March 2004, Lancaster, UK.

148. LABORATORY OF INTEGRATED SYSTEMS. Polytechnic School, University of São Paulo, Brazil. http://www.lsi.usp.br/_reeng2/ing/index.htm

149. LADDAD, R. I want my AOP!. **Java World Magazine**. January, March and April 2002.

150. LAI, A.; MURPHY, G.; WALKER, R. Separating Concerns with Hyper/J: An Experience Report. **Proceedings of the International Workshop on Multi-Dimensional Separation of Concerns in Software Engineering at ICSE**, 2000.

151. LANG, J.; STEWART, D. A Study of the Applicability of Existing Exception-Handling Techniques to Component-Based Real-Time Software Technology, **ACM Computing Surveys** 20, 274-301 (1998).

152. LANGE, D.; OSHIMA, M. **Programming and Deploying Java Mobile Agents with Aglets**. Addison-Wesley, 1998.

153. LAVENDER, R.; SCHMIDT, D. Active Object: an Object Behavioral Pattern for Concurrent Programming. In: **Pattern Languages of Program Design** (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), (Reading, MA), Addison-Wesley, 1996.

154. LEA, D. **Concurrent Programming in Java: Design Principles and Patterns**. 2nd Edition, Addison-Wesley, November 1999.

155. LEHMAN, T.; MCLAUGHRY, S.; WYCKOFF, P. TSpaces: The Next Wave. **Proceedings of the Hawaii International Conference on System Sciences (HICSS-32)**, January 1999.

156. LESIECKI, N. Test Flexibility with AspectJ and Mock Objects. **Java Technology Zone for IBM's Developer Works**, May 2002.

157. LETIER, E.; LAMSWEERDE, A. Agent-based Tactics for Goal-Oriented Requirements Elaboration. **Proceedings of 24th International Conference on Software Engineering**, Toronto, Canada, May 2001.

158. LI, W.; HENRY, S. Object-Oriented Metrics that Predict Maintainability. **Journal of Systems and Software**, 23 (2), February 1993, pp. 111-122.

159. LIEBERHERR, K. Adaptive **Object-Oriented Software: The Demeter Method with Propagation Patterns**. PWS Publishing Company, Boston, 1996.

160. LIEBERHERR, K.; ORLEANS, D.; OVLINGER, J. Aspect-Oriented Programming with Adaptive Methods. **Communications of the ACM**, 44(10):39–41, October 2001.

161. LIPPERT, M.; LOPES, C. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. **Proceedings of the 22nd International Conference on Software Engineering**, Limerick, Ireland, may 2000, pp. 418 – 427.

162. LOPES, C. **D: A Language Framework for Distributed Programming**. PhD Thesis, College of Computer Science, Northeastern University, 1997.

163. LOPES, C. **Aspect-Oriented Programming: An Historical Perspective (What's in a Name?)**. ISR Technical Report #UCI-ISR-02-5, University of California, Irvine, December 2002.

164. LUCENA, C.; GARCIA, A.; ROMANOVSKY, A.; CASTRO, J.; ALENCAR, P. (editors). **Software Engineering for Multi-Agent Systems II**. Lecture Notes in Computer Science, vol. 2940, Springer-Verlag, February 2004.

165. MAES, P. Concepts and Experiments in Computational Reflection. **Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'87)**, ACM SIGPLAN Notices, October (1987): 147-155.

166. MAES, P. Agents that Reduce Work and Information Overload. **Communications of the ACM**, 37 (7), 31-40, 1994.

167. MALONE, T.; CROWSTON, K. The Interdisciplinary Study of Coordination. **ACM Computing Surveys**, 26, 1 (March 1994), pp. 87-119.

168. MCGARRY, F.; PAJERSK, R.; PAGE, G.; WALIGORA, S.; BASILI, V.; ZELKOWITZ, M. **Software Process Improvement in the NASA Software Engineering Laboratory.** Carnegie Mellon Univ., Software Eng. Inst., Technical Report CMU/SEI-95-TR-22, December 1994.

169. MESSAGE WEBSITE, Available at URL http://www.eurescom.de/Public/Projects/p900-series/P907/P907.htm.

170. MEYER, B. **Object-Oriented Software Construction**. 2.ed. Prentice Hall, 1997.

171. MINSKY, N.; UNGUREANU, V. Law-Governed Interaction: A Coordination and Control Mechanism for Heterogeneous Distributed Systems. **ACM Transactions on Software Engineering and Methodology**, Vol. 9, No. 3, July 2000, pp. 273-305.

172. MITCHELL, T. **Machine Learning**. McGraw Hill, New York, 1997.

173. MOFFAT, D.; FRIJDA, N. Where There's a Will There's an Agent. In: Wooldridge, M., Jennings, N. (Eds.), **Intelligent Agents: Theories, Architectures and Languages**, LNAI 890: Springer Verlag, pp. 245-260, 1995.

174. MOSS, S. et al. SDML: A Multi-Agent Language for Organizational Modelling. **Computational Mathematical Organization Theory**, Vol. 4, n. 1, (1998): 43-69.

175. MURPHY, G. et al. Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-Oriented Programming. **IEEE Transactions on Software Engineering**, Special Section on Empirical Software Engineering, July/August 1999.

176. MURPHY, G. et al. Does aspect-oriented programming work?. **Communications of the ACM**, pp. 75-77, Issue 10, Vol. 44, October 2001.

177. NAGEL, K. **High-speed Microsimulation of Traffic Flow**. PhD Thesis, University of Cologne, Germany, 1995.

178. NAGEL, K.; SCHRECKENBERG, M. **A Cellular Automaton Model for Freeway Traffic**. Journal de Physique I, France 2, 2221 (1992).

179. NODINE, M.; PERRY, B.; UNRUH, A. Experience with the InfoSleuth Agent Architecture. **Proceedings of the AAAI-98 Workshop on Software Tools for Developing Agents**, 1998.

180. NORMAN, T.; LONG, D. Goal Creation in Motivated Agents. In: Wooldridge, M., Jennings, N. (Eds.), **Intelligent Agents: Theories, Architectures, and Languages**, LNAI 890: Springer Verlag, 1995.

181. NWANA, H. et al. ZEUS: An Advanced Toolkit for Engineering Distributed Multi-Agent Systems. **Applied Artificial Intelligence Journal**, 1999, 13(1):129-186.

182. NWANA, H. Software Agents: An Overview. **Knowledge Engineering Review**, September 1996, 11(3): 1-40.

183. OBJECT MANAGEMENT GROUP – AGENT PLATFORM SPECIAL INTEREST GROUP. **Agent Technology – Green Paper**. Version 1.0, September 2000.

184. ODELL, J.; PARUNAK, H.; BAUER, B. Extending UML for Agents. **Proceedings of the Agent-Oriented Information Systems Workshop**, 17th National Conference on Artificial Intelligence, J. Odell, H. Parunak, B. Bauer (eds.), 2000.

185. OHSUGA, A. et al. Plangent: An Approach to Making Mobile Agents Intelligent. **IEEE Internet Computing**, 1(4), July 1997.

186. OLIVA, A.; BUZATO, L. Design and Implementation of Guaraná. **Proceedings of the Conference on Object-Oriented Technologies and System (COOTS)**, May 1999.

187. OMICINI, A.; ZAMBONELLI, F. TuCSoN: a Coordination Model for Mobile Information Agents. **Proceedings of the 1st International Workshop on Innovative Internet Information Systems (IIIS'98)**, Pisa (I), June 1998.

188. OSSHER, H.; TARR, P. Using Multi-dimensional Separation of Concerns to (Re)Shape Evolving Software. **Communications of the ACM**, 44(10):43–50, October 2001.

189. PACE, A.; CAMPO, M.; SORIA, A. Architecting the Design of Multi-Agent Organizations with Proto-Frameworks. **Software Engineering for Large-Scale Multi-Agent Systems**, C. Lucena et al (eds.). Springer-Verlag, LNCS 2940, Berlin, February 2004, pp. 75-92.

190. PACE, A.; TRILNIK, F.; CAMPO, M. Assisting the Development of Aspect-based MAS using the SmartWeaver Approach. **Software Engineering for Large-Scale Multi-Agent Systems**, A. Garcia et al (eds.). Springer-Verlag: LNCS 2603, Berlin, April 2003.

191. PAPAIOANNOU, T.; EDWARDS, J. Building Agile Systems with Mobile Code. **Journal of Autonomous Agents and Multi-Agent Systems**, vol. 4, no. 4, 2001, pp. 293-310.

192. PAPURT, D. The Use of Exceptions. **Journal of Object-Oriented Programming**, 13-17 (1998).

193. PARNAS, D. On the Criteria to be Used in Decomposing Systems into Modules. **Communications of the ACM**, 15 (12), December 1972, pp. 1053-1058.

194. POGGI A. DAISY: An Object-Oriented System for Distributed Artificial Intelligence. **Intelligent Agents**, 1995, LNAI 890, pp.341-354.

195. POPOVICI, A.; GROSS, T.; ALONSO, G. Dynamic Weaving for Aspect Oriented Programming. **Proceedings of the 1st International Conference on Aspect-Oriented Software Development**, Enschede, The Netherlands, April 2002.

196. POSLAD, S.; BUCKLE P.; HADINGHAM, R. The FIPA-OS Agent Platform: Open Source for Open Standards. **Proceedings of the 5th International Conference on the Practical Application of Intelligent Agents and Multi-Agents**, Manchester, UK, April 2000, pp. 355-368.

197. RAO, A.; GEORGEFF, M. Modelling Rational Agents within a BDI Architecture. **Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning**, Cambridge, MA, 1991; 473-484.

198. RAO, A.; GEORGEFF, M. BDI Agents: From Theory to Practice. **Proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS-95)**, San Francisco, 1995; 312-319.

199. RASHID, A. A Hybrid Approach to Separation of Concerns: The Story of SADES. **Proceedings of the 3rd International Conference on Meta-Level Architectures and Separation of Concerns (Reflection 2001)**, Lecture Notes in Computer Science 2192, pp. 231-249

200. RASHID, A.; MOREIRA, A.; ARAUJO; J. Modularisation and Composition of Aspectual Requirements. **Proceedings of the 2nd International Conference on Aspect-Oriented Software Development**, ACM Press, 2003, pp. 11-20.

201. RASHID, A.; SAWYER, P.; MOREIRA, A.; ARAUJO; J. Early Aspects: A Model for Aspect-Oriented Requirements Engineering. **Proceedings of the IEEE Joint International Conference on Requirements Engineering**, 2002, IEEE Computer Society Press, pp. 199-202.

202. RASMUS, D. **Rethinking Smart Objects: Building Artificial Intelligence with Objects**. Cambridge University Press, New York, 1999.

203. RIEHLE, D. Composite Design Patterns. **Proceedings of the 1997 Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA '97)**, ACM Press, Page 218-228, 1997.

204. RIEHLE, D. Bureaucracy. In: **Pattern Languages of Program Design 3**, R. Martin, D. Riehle, F. 35 (Ed.), Addison Wesley, 1998, pp. 163 - 185.

205. RIPPER, P.; FONTOURA, M.; NETO, A.; LUCENA, C. V-Market: A Framework for e-Commerce Agent Systems. **World Wide Web**, 2000, Baltzer Science Publishers, 3(1): 43-52.

206. ROBILLARD, M.; MURPHY, G. FEAT: A Tool for Locating, Describing, and Analyzing Concerns in Source Code. **Proceedings of the International Conference on Software Engineering (ICSE'03)**, Portland, May 2003, pp. 822-824.

207. RUSSELL, S.; NORVIG, P. **Artificial Intelligence: A Modern Approach**. Prentice Hall, 2nd edition, 2002.

208. RYMAN, A. **The Theory-Model Paradigm in Software Design**. Technical Report TR 74.048, IBM Corporation (October 1989).

209. SALINGAROS, N. The Structure of Pattern Languages. **Architectural Research Quarterly**, volume 4**,** Cambridge University Press, 2000, pp. 149-161.

210. SANT'ANNA, C.; GARCIA, A.; CHAVEZ, C. ; LUCENA, C.; VON STAA, A. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. **Proceedings of 17th Brazilian Symposium on Software Engineering**, Manaus, Brazil, October 2003.

211. SARDINHA, J.; MILIDIÚ, R.; LUCENA, C. Engineering Machine Learning Techniques into Multi-Agent Systems. Submitted to **International Journal of Software Engineering and Knowledge Engineering**, 2004.

212. SARDINHA, J.; RIBEIRO, P.; LUCENA, C.; MILIDIÚ, R. An Object-Oriented Framework for Building Software Agents. **Journal of Object Technology**. January - February 2003, Vol. 2, No. 1.

213. SHAVOR, S.; D'ANJOU, J.; FAIRBROTHER, S. **The Java Developer's Guide to Eclipse**. Addison-Wesley, 2003.

214. SHAW, M.; GARLAN, D. **Software Architecture: Perspectives on an Emerging Discipline**. Prentice Hall (1996).

215. SCHIAFFINO, S.; AMANDI, A. On the Design of a Software Secretary. **Proceedings of the Argentine Symposium on Artificial Intelligence (ASAI'02)**, 2002, pp. 218-230.

216. SHOHAM, Y. Agent0 - A Simple Agent Language and its Interpreter. **Proceedings of the Ninth National Conference on Artificial Intelligence**, 1991, pp. 704–709.

217. SHOHAM, Y. Agent-Oriented Programming. **Artificial Intelligence**. v.60, n.1, p.51-92, Mar. 1993.

218. SILVA, O. et al. A Shared-Memory Agent-Based Framework for Business-to-Business Applications. **Proceedings of the 2001 Information Resources Management Association International Conference (IRMA'2001)**, May 2001.

219. SILVA, O.; GARCIA, A.; LUCENA, C. T-Rex: A Reflective Tuple Space Environment for Dependable Mobile Agent Systems. **Proceedings of the Workshop on Wireless Communication and Mobile Computation (WCSF'2001) at 3rd. IEEE Int'l Conference on Mobile and Wireless Communication Networks**, Recife, Brazil, August 2001, pp. 1-10.

220. SILVA, O.; GARCIA, A.; LUCENA, C. A Unified Software Architecture for System-Level and Agent-Level Dependability in Multi-Agent Object-Oriented Systems. **Proceedings of the 7th ECOOP Workshop on Mobile Objects Systems**, Budapest, Hungary, June 2001.

221. SILVA, O.; GARCIA, A.; LUCENA, C. The Reflective Blackboard Pattern: Architecting Large-Scale Multi-Agent Systems. In: Garcia, A. et al (Eds).

Software Engineering for Large-Scale Multi-Agent Systems, Springer-Verlag, LNCS 2603, April 2003, pp. 73-93.

222. SILVA, V. **From a Conceptual Framework for Agents and Objects to a Multi-Agent System Modeling Language**. PhD Thesis, Computer Science Department, PUC-Rio, March 2004.

223. SILVA, V.; GARCIA, A.; BRANDÃO, A.; CHAVEZ, C.; LUCENA, C.; ALENCAR, P. Taming Agents and Objects in Software Engineering. In: **Software Engineering for Large-Scale Multi-Agent Systems**, A. Garcia, C. Lucena, J. Castro, A. Omicini, F. Zambonelli (eds.). Springer-Verlag: LNCS 2603, Berlin, April 2003.

224. SKIPPER, M. The Watson Subject Compiler & AspectJ: A Critique of Practical Objects. **Proceedings of the Workshop on Multi-Dimensional Separation of Concerns at OOPSLA**, 1999.

225. SMITH, B. Reflection and Semantics in Lisp. **Proceedings of the ACM Symposium on Principles of Programming Languages**, pages 23:35, Salt Lake City, UT, January 1984. ACM Press.

226. SOARES, S.; LAUREANO, E.; BORBA, P. Implementing Distribution and Persistence Aspects with AspectJ. **Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)**, pp. 174-190.

227. SOC+AGENTS GROUP. **Separation of Concerns and Multi-Agent Systems**. URL: http://www.teccomm.les.inf.puc-rio.br/227/

228. SOMMERVILLE, I. **Software Engineering**. 6.ed. Harlow, England, Addison-Wesley, 2001.

229. SOUZA, G.; SILVA, I.; CASTRO, J. Adapting the NFR Framework to Aspect-Oriented Requirements Engineering. **Proceedings of the 17th Brazilian Symposium on Software Engineering**, Manaus, Brazil, October 2003.

230. SPURLIN, V. Aspect-Oriented Programming with Sun ONE Studio. In **Sun ONE Studio Developer Resource page**. October 2002. http://forte.sun.com/ffj/articles/aspectJ.html

231. SUTTON, S.; ROUVELLOU, I. Modeling of Software Concerns in Cosmos. **Proceedings of the 1st International Conference on Aspect-Oriented Software Development**, 2002, ACM Press, pp. 127-133.

232. SYCARA, K. et al. The RETSINA MAS Infrastructure. **Journal of Autonomous Agents and Multi-Agent Systems**, July 2003, 7(1-2).

233. TAIVALSAARI, A. On the Notion of Inheritance. **ACM Computing Surveys**, 28(3):438–479, September 1996.

234. TARR, P. et al. N Degrees of Separation: Multi-Dimensional Separation of Concerns. **Proceedings of the 21st International Conference on Software Engineering**, May 1999, pp. 107–119.

235. TARR, P., OSSHER, H. **Hyper/J User and Installation Manual**, 2000. http://www.alphaworks.ibm.com/tech/hyperj

236. Technology Review 10 (Ten Emerging Technologies That Will Change the World). **Technology Review**, 104(1):97, 113, January/February 2001.

237. THE ECLIPSE WEBSITE. URL: http://www.eclipse.org.

238. ASPECTJ PROJECT, 2003. http://www.eclipse.org/aspectj/.

239. TRESE GROUP. http://trese.cs.utwente.nl/taosad/

240. UBAYASHI, N.; TAMAI, T. Separation of Concerns in Mobile Agent Applications. **Proceedings of the 3rd International Conference Reflection 2001**, LNCS 2192, Kyoto, Japan, September 2001, Springer, pp. 89-109.

241. VANHILST, M.; NOTKIN, D. Using Role Components to Implement Collaboration- Based Designs. **Proceedings of the 1996 Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA'96)**, ACM Press, 1996, pp. 359 - 369.

242. VAN LAMSWEERDE, A. Goal-Oriented Requirements Engineering: A Guided Tour. **Proceedings of the International Joint Conference on Requirements Engineering**, Toronto, IEEE, August 2001, pp.249-263.

243. VAN SPLUNTER, S.; WIJNGAARDS, N.; BRAZIER, F. Structuring Agents for Adaptation. In: Alonso, E. et al (Eds), **Adaptive Agents and Multi-Agent Systems**, LNAI, Vol. 2636, 2003, pp. 174-186.

244. WAGNER, G. Agent-Object-Relationship Modeling. **Proceedings of the 2nd International Symposium: From Agent Theory to Agent Implementation**, April 2000.

245. WALKER, R.; BANIASSAD, E.; MURPHY, G. An Initial Assessment of Aspect-Oriented Programming.**Proceedings of the 21st International Conference on Software Engineering**, pp. 120-130, 1999.

246. WEERASOORIYA, D.; RAO, A.; RAMAMOHANARAO, K. Design of a Concurrent Agent-Oriented Language. In: Wooldridge, M.; Jennings, N. (Eds.). **Intelligent Agents**. Berlin: Springer-Verlag, 1995. p.386-401. (LNAI, v. 890).

247. WEISS, G. (Editor). **Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence**. The MIT Press, 1999.

248. WOOLDRIDGE, M.; CIANCARINI, P. Agent-Oriented Software Engineering: The State of the Art. In: P. Ciancarini and M. Wooldridge (Eds.), **Agent-Oriented Software Engineering**, Springer-Verlag, LNAI 1957, Berlin, January 2001.

249. WOOLDRIDGE, M.; JENNINGS, N.; KINNY, D. The Gaia Methodology for Agent-oriented Analysis and Design. **Journal of Autonomous Agents and Multi-Agent Systems**, Vol. 3, 2000, pp. 285–312.

250. YOSHIOKA, H.; TAHARA, Y.; OHSUGA, A.; HONIDEN, S. Security for Mobile Agents. **Proceedings of the 1st International Workshop on Agent-Oriented Software Engineering at ICSE 2000**, Limerick (IR), June 2000.

251. YU, L.; SCHMID, B. A Conceptual Framework for Agent-Oriented and Role-Based Work on Modeling. In: Wagner, G., Yu, E. (eds.): **Proceedings of the 1st Int. Workshop on Agent-Oriented Information Systems** (1999).

252. ZACARIA, A., HOSNY, H. Metrics for Aspect-Oriented Software Design. **Proceedings of the 3rd International Workshop on Aspect-Oriented Modeling**, AOSD'03 Conference, 2003.

253. ZAMBONELLI, F.; JENNINGS, N.; WOOLDRIDGE, M.: Organizational Abstractions for the Analysis and Design of Multi-agent Systems. In: Ciancarini, P., Wooldridge, M. (eds.): **Agent-Oriented Software Engineering**, Springer-Verlag (2001).

254. ZHAO, J. **Towards a Metrics Suite for Aspect-Oriented Software**. Technical-Report SE-136-25, Information Processing Society of Japan (IPSJ), March 2002.

255. REFAZENDA WEBSITE. **Refactoring with Aspects**. URL: http://twiki.im.ufba.br/bin/view/Aside/Refazenda

256. FOWLER, M. et al. **Refactoring: Improving the Design of Existing Code**. Addison Wesley, 1999.

257. GUESSOUM, Z. et al. Dynamic and Adaptive Replication for Large-Scale Reliable Multi-agent Systems. In: **Software Engineering for Large-Scale Multi-Agent Systems**, A. Garcia et al (eds.). Springer-Verlag: LNCS 2603, Berlin, April 2003, pp. 182-198.

258. ALENCAR, P.; BARRENECHEA, E.; GARCIA, A.; LUCENA, C.; COWAN, D. **A Query-Based Approach for Aspect Understanding, Measurement and Analysis**. Technical Report CS-2004-13, School of Computer Science, University of Waterloo, Canada, February 2004. (Submitted to IEEE Transactions on Software Engineering)

259. SILVA, O.; GARCIA, A.; LUCENA, C. The Reflective Blackboard Architectural Pattern for Developing Large Scale Multi-Agent Systems. **Proceedings of the 1st International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2002) at ICSE 2002**, Orlando, USA, May 2002.

260. CHAVEZ, C.; GARCIA, A.; LUCENA, C. **From AOP to MDSOC: An Experience Report**. Technical Report, Computer Science Department, PUC-Rio, Brazil, September 2001.

261. ODELL, J.; PARUNAK, H.; FLEISCHER, M. The Role of Roles in Designing Effective Agent Organizations. **Software Engineering for Large-Scale Multi-Agent Systems**, Lecture Notes in Computer Science, vol. 2603, Springer-Verlag, April 2003, pp. 27-38.

262. ODELL,J. Objects and Agents Compared. **Journal of Object Technology**, vol. 1, n. 1, May 2002, pp. 41-53. www.jot.fm/issues/issue_2002_05.

263. CASTRO, J.; KOLP, M.; MYLOPOULOS, J. Towards Requirements-Driven Information Systems Engineering: The Tropos Project. **Information Systems** 27(6): 365-389 (2002).

264. SANT'ANNA, C. **Manutenibilidade e Reusabilidade de Software Orientado a Aspectos: Um Framework de Avaliação**. Master's Dissertation, Computer Science Department, PUC-Rio, March 2004. (In Portuguese)

265. KULESZA, U.; GARCIA, A.; SANT'ANNA, C.; VON STAA, A.; LUCENA, C. **Avaliação de um Conjunto de Métricas para Desenvolvimento Orientado a Aspectos**. Technical Report, Computer Science Department, PUC-Rio, Brazil, March 2004. (In Portuguese)

266. CHAVEZ, C.; GARCIA, A.; LUCENA, C. **From AOP to MDSOC: An Experience Report**. Technical Report, Computer Science Department, PUC-Rio, Brazil, September 2001.

267. LAMB, D. A. **Software Engineering: Planning for Change**. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

Apêndice I
Implementação da linguagem de padrões: Problemas e Código de amostra em AspectJ

Este apêndice descreve a implementação do sistema Expert Committee (EC) (Seção 5.1) usando a linguagem de padrões proposta (Capítulo 5) e a linguagem AspectJ, versão 1.3. A descrição inclui partes de código relevantes para a compreensão de como implementar os padrões propostos. Além disso, avaliamos a facilidade com que essa linguagem oferece suporte à definição dos padrões propostos.

O padrão Kernel do Agente

Etapa 1: Como definir uma classe básica do agente?

A classe `Agent` é declarada como abstrata (linha 1) uma vez que precisa ser estendida para diferentes tipos de agente. Os tipos de agente específicos são definidos como subclasse da classe `Agent` (etapa 5). A classe `Agent` deve definir atributos genéricos, independente do tipo de agente. Por exemplo, a classe `Agent` no sistema ExpertCommittee (EC) inclui os atributos a seguir:

- um identificador (linha 3);
- um hash que associa um par ação-performativa (chave de hash) com um objetivo reativo (linha 4);
- um hash que associa um objetivo reativo específico (chave de hash) com uma lista de planos que permite alcançar esse objetivo (linha 5);
- uma lista de planos que devem ser realizados (linha 6);
- uma lista de objetivos que devem ser alcançados (linha 7);
- um hash que mantém identificadores de agente (chave de hash) e as informações de agente correspondentes, como local, recursos (linha 8).

Ademais, a classe `Agent` também deve definir os métodos básicos para a manipulação de suas crenças, objetivos e planos (linhas 11-35). Também pode implementar serviços gerais como métodos. Todos os tipos de agente herdarão a implementação desses serviços gerais.

```
1.    public abstract class Agent {
2.
3.          protected String identifier;
4.          protected Hashtable reactiveGoalList;
5.          protected Hashtable reactivePlanList;
6.          protected List toPerformPlans;
7.          protected Vector toAchieveGoals;
8.          protected Hashtable agentList;
9.          ...
10.
11.         public void init() {
12.               this.reactivePlanList = new Hashtable();
13.               this.toAchieveGoals = new Vector();
14.               this.toPerformPlans = new Vector();
15.               this.reactiveGoalList = new Hashtable();
16.               this.agentList = new Hashtable();
17.         }
18.
19.         public List getToPerformPlans() {
20.               return this.toPerformPlans;
21.         }
22.
24.         public  void setPlan(Plan newPlan) {
25.               this.toPerformPlans.addElement(newPlan);
26.         }
27.
28.         public  void removePlan(Plan plan) {
29.               this.toPerformPlans.removeElement(plan);
30.         }
31.
32.         public Hashtable getAgentList() {
33.               return agentList;
34.         }
35.         ...
36.
37. }
```

Etapa 2: Como estruturar (ou definir) classes da crença?

As classes Belief são definidas como classes do domínio. Cada crença é declarada como um atributo: (i) na classe `Agent` (etapa 1) quando a crença é necessária para cada tipo de agente; ou (ii) nas subclasses Agent (etapa 5) – quando é necessário apenas em um tipo de agente específico. Por exemplo, a classe `Agent` do

sistema EC define uma crença geral para manter informações sobre agentes no ambiente (linha 8).

Etapa 3: Como estruturar (ou definir) classes do objetivo?

As classes Goal são estruturadas como uma hierarquia de classes. A hierarquia organiza-se da seguinte forma:

- a raiz da hierarquia é a classe abstrata `Goal`;
- As classes `ReactiveGoal`, `ProactiveGoal` e `DecisionGoal` herdam da classe abstrata `Goal`, representando o segundo nível da hierarquia;
- finalmente, definimos as classes do objetivo específicas à aplicação definindo subclasses das classes `ReactiveGoal`, `ProactiveGoal` e `DecisionGoal`.

A classe abstrata `Goal` define o atributo de nome . Além disso, pode definir uma lista de subobjetivos com os quais o objetivo é associado. A classe `Goal` também define dois métodos abstratos: `changeBelief()` e `eval()`.

Etapa 4: Como estruturar (ou definir) classes do plano?

A classe `Plan` é declarada como abstrata (linha 1) uma vez que precisa ser estendida para definir planos específicos a aplicações. Cada plano é definido para alcançar um objetivo específico. Ele define os campos a seguir: (i) `agent` – uma referência à instância `Agent` que está executando o plano (linha 3); (ii) `goal` – uma referência à instância `Goal` que representa o objetivo associado ao plano (linha 4); (iii) `preConditions` – a lista de precondições para a execução de planos (linha 6); e (iv) `posConditions` – a lista de pós-condições para a execução de planos (linha 7).

A classe `Plan` também define alguns métodos abstratos, como: (i) `execute()` – usado para definir as ações internas do plano a fim de cumprir com os objetivos atribuídos; (ii) `stop()` – usado para interromper o plano; (iii) `checkPreConditions()` – verifica se as precondições foram satisfeitas; e (iv)

`checkPosConditions()` – verifica se as pós-condições foram satisfeitas. As subclasses `Plan` implementam esses métodos a fim de definir os planos específicos a aplicações.

Em linguagem Java, a classe `Plan` deve incorporar recursos de thread ao implementar a interface `Runnable` (linha 1). Nesse caso, o método `run()`, que define a execução da thread, deve chamar o método abstrato `execute()` (linha 12-14). O padrão Autonomia (Seção 5.5) especifica a estratégia de concorrência que permite a execução simultânea de vários planos.

```
1.    public abstract class Plan implements Runnable {
2.
3.          public Agent agent;
4.          public Goal goal;
5.
6.          public Vector preConditions = new Vector();
7.          public Vector posConditions = new Vector();
8.
9.          public abstract void execute(Agent agent, Goal goal);
10.         public abstract void stop();
11.
12.         public void run(){
13.               this.executePlan(this.agent, this.planGoal);
14.         }
15.
16.         public abstract boolean checkPreConditions(Agent agent);
17.
18.         public abstract boolean checkPosConditions(Agent agent);
19.         ...
20.   }
```

Etapa 5: Como definir tipos de agente específicos?

Os tipos de agente são definidos como subclasse da classe `Agent` (etapa 1). Eles declaram as crenças específicas a tipos como atributos. Ademais, também definem métodos para manipular essas crenças e implementar ações específicas a tipos.

No sistema EC, definimos a classe `ResearcherUserAgent`, uma subclasse de `Agent`. Como essa classe é um agente de usuário que representa os pesquisadores no sistema, ela precisa definir crenças específicas (linhas 3-7) relacionadas ao pesquisador, como, nome de usuário, interesses da pesquisa, posição atual, etapa da

pesquisa, pauta etc. Além disso, métodos específicos precisam ser definidos para tratar suas crenças (linha 9-25).

Ao definir tipos de agente específicos, é comum especificar novas subclasses do plano e objetivo. No sistema EC, definimos novos objetivos e planos para os papéis exercidos pelo agente de usuário do pesquisador.

```
1.    public class ResearcherUserAgent extends Agent {
2.
3.          protected String userName;
4.          protected Hashtable researchInterests;
5.          protected String currentPosition;
6.          protected String researchStage;
7.          protectend Agenda agenda;
8.
9.          ...
10.         public String getUserName() {
11.               return userName;
12.         }
13
14.         public Hashtable getResearcherInterests() {
15.               return researchInterests;
16.         }
17.
18.         public void setUserName(String userName) {
19.               this.userName = userName;
20.         }
21.
22.         public void setResearchInterests(Hashtable
interestTable) {
23.               this.researchInterests = interestTable;
24.         }
25.         ...
26.   }
```

O padrão Interação

Etapa 1: Como definir uma classe de interação abstrata?

O aspecto `Interaction` é declarado como abstrato (linha 1) uma vez que precisa ser redefinido em diferentes contextos, ou seja, diferentes tipos de agente e papéis. O aspecto define diferentes atributos, como `inbox` (linha 2), `outbox` (linha 3), `sensors` (linha 4) e `effectors` (linha 5). Métodos de atualização também são definidos, por exemplo, para atualizar a caixa de entrada e de saída (linhas 7-14).

```
1. public abstract aspect Interaction {
2.    private Vector Agent.inBox = new Vector();
3.    private Vector Agent.outBox = new Vector();
4.    private Vector Agent.sensors = new Vector();
5.    private Hashtable Agent.effectors = new Hashtable();
6.
7.    private void Agent.updateInBox(Message msg) {
8.          inBox.addElement(msg);
9.    }
10.
11.   private void Agent.updateOutBox(Message msg) {
12.         outBox.addElement(msg);
13.   }
14.   ...
```

Cada instância de agente deve ter sua própria caixa de entrada, saída, sensores e efetores. Como conseqüência, o aspecto `Interaction` deve ser instanciado por instância Agent (padrão Kernel). A versão atual de AspectJ oferece suporte à especificação de aspectos por-objeto. Em nosso caso, descrevemos a instanciação do aspecto `Interaction` usando `perthis`:

```
public abstract aspect Interaction perthis(Agent) {…}
```

Entretanto, o uso de `perthis` restringe o escopo do aspecto. Quando um aspecto de AspectJ é declarado como único ou estático, seu escopo é todo o sistema e o aspecto pode afetar todas as classes do sistema. Os aspectos por-objetos só podem afetar o objeto com o qual está associado. Como o protocolo de interação afeta várias classes, nem só a classe `Agent`, a condição `perthis` não pode ser usada nesse contexto. Como resultado, tivemos de declarar nosso aspecto como único e introduzir os métodos e atributos (linhas 2-14) à classe `Agent`. Observe que apesar de a estrutura do padrão Interação não descrever esses membros do aspecto como parte de uma interface de crosscutting, eles tiveram de ser introduzidos em nossa implementação de AspectJ devido a restrições de AspectJ. São declarados como particular, o que significa que “eles são particulares do aspecto”: somente o código no aspecto pode ver esses campos e métodos. Se a classe `Agent` só tiver membros particulares chamados da mesma forma (declarado em `Agent` ou em outro aspecto), não haverá uma colisão de nome, uma vez que não haverá referências ambíguas a esses membros.

O uso de inter-type declarations complicou o projeto do aspecto `Interaction` uma vez que requer que a instância do agente seja exposta como um parâmetro em cada advice do aspecto `Interaction`. Nosso princípio de projeto consiste em introduzir métodos apenas quando são interceptados ou modificados por outros aspectos. Do nosso ponto de vista, as classes devem estar cientes dos aspectos. Mesmo o comportamento introduzido pelos aspectos não deve ser acessado diretamente pelas classes porque aumenta o acoplamento do sistema. Essa decisão de projeto pode ser evitada usando pointcuts e advices que conectam aspectos e classes [115].

```
15.   private void Agent.receiveMsg(Message msgUnmarshalled) {
16.         updateInBox(msgUnmarshalled);
17.   }
18.   public void sendMsg(Message msg, Agent agent){
19.         agent.updateOutBox(msg);
20.         agent.sendMsgEffector(msg, agent);
21.   }
22.
23.   public abstract void sendMsgEffector(Message msg, Agent
agent);
24.   public void init(Agent agent) {
25.         agent.inBox = new Vector();
26.         agent.outBox = new Vector();
27.         agent.sensors = new Vector();
28.         agent.effectors = new Hashtable();
29.
30.         // Init sensors and effectors
31.         initSensorsAndEffectors(agent);
32.
33.         // Set the agent reference in every sensor
34.         Enumeration sensorsEnum = (agent.sensors).elements();
35.         Sensor sensor;
36.         while (sensorsEnum.hasMoreElements()) {
37.               sensor = (Sensor) sensorsEnum.nextElement();
38.               sensor.setAgent(agent);
39.         }
40.   }
41.
42.   public abstract void initSensorsAndEffectors(Agent agent);
```

O aspecto `Interaction` define dois métodos fundamentais:

- `receiveMsg()` – este método (linha 15-17) é chamado pelos advices do aspecto responsáveis pelo recebimento de mensagens externas do ambiente (sensores do agente ou classes específicas a domínios);

- `sendMsg()` – este método (linha 18-21) é chamado pelos advices do aspecto responsáveis pelo envio de mensagens para o ambiente. O código-fonte mostra que esse método chama o método abstrato `sendMsgEffector()` (linha 23). Esse método abstrato implementa a lógica para decidir qual efetor do agente deve ser usado para enviar uma determinada mensagem. Os subaspectos Interaction devem implementar esse método abstrato.

Além disso, o aspecto `Interaction` define pointcuts e advices a fim de: (i) inicializar os recursos de interação do agente; (ii) definir a interface de crosscutting IncomingMessage; e (iii) definir a interface crosscutting OutgoingMessage. Descrevemos a seguir (etapas 2, 3 e 4) a implementação desses elementos.

Etapa 2: Como inicializar os recursos de interação do agente?

A inicialização dos recursos de interação requer a definição do pointcut abstrato `agentInstantiation()` (linha 61). Esse pointcut define o join point na execução do papel ou agente onde é inicializado o aspecto `Interaction`. Em geral, ele age nos construtores de classes do agente específicas. Os subaspectos Interaction (etapa 5) devem implementar esse pointcut abstrato. A inicialização de Interaction também precisa da definição de um advice acoplado ao pointcut `agentInstantiation()`. Deve ser definida como um after advice (linhas 62-64) para chamar o método `init()` do aspecto Interaction.

O método `init()` inicializa os atributos de interação, como estruturas de dados para armazenar mensagens (caixa de entrada e caixa de saída), sensores e efetores (linhas 43-59). Ademais, também chama o método abstrato

`initSensorsAndEffectors()` (linha 42). Esse método inicializa sensores e efetores usados por um tipo específico de papel do agente. Ele precisa ser implementado pelos subaspectos Interaction (etapa 5).

```
40.   }
41.
42.   public abstract void initSensorsAndEffectors(Agent agent);

43.   public void init(Agent agent) {
44.         agent.inBox = new Vector();
45.         agent.outBox = new Vector();
46.         agent.sensors = new Vector();
47.         agent.effectors = new Hashtable();
48.
49.         // Init sensors and effectors
50.         initSensorsAndEffectors(agent);
51.
52.         // Set the agent reference in every sensor
53.         Enumeration sensorsEnum = (agent.sensors).elements();
54.         Sensor sensor;
55.         while (sensorsEnum.hasMoreElements()) {
56.               sensor = (Sensor) sensorsEnum.nextElement();
57.               sensor.setAgent(agent);
58.         }
59.   }
60.
61.   protected abstract pointcut agentInstantiation(Agent agent);


62.   after(Agent agent) : agentInstantiation(agent) {
63.         init(agent);
64.   }
```

Etapa 3: Como definir uma interface crosscutting IncomingMessage?

A interface crosscutting IncomingMessage define como interceptar sensores e classes específicos a domínios para receber mensagens ou detectar eventos externos. Ele define dois pointcuts no aspecto `Interaction`: (i) o pointcut `incomingMsgSensor()` (linha 65-67), que intercepta todas as subclasses de `Sensor`; e (ii) o pointcut abstrato `incomingMsgDomainSpecific()` (linha 77-78), que intercepta as classes específicas a domínios. O último precisa ser implementado pelos subaspectos Interaction (etapa 5).

Os pointcuts são associados a after advices. Esses advices chamam o método `receiveMsg()` em uma instância do agente específica (linhas 69-75).

```
65.   pointcut incomingMsgSensor(Sensor sensor, Message msg):
66.         (this(sensor) && args(msg) &&
67.          execution(void
Sensor.receiveUnmarshalledMsg(Message)));
68.
69.   after(Sensor sensor, Message msg):
70.               incomingMsgSensor(sensor, msg){
71.         Agent agent = sensor.getAgent();
72.         if (msg != null) {
73.               agent.receiveMsg(msg);
74.         }
75.   }
76.
77.   protected abstract pointcut incomingMsgDomainSpecific
78.                            (Agent agent, Object
interceptedObject);
```

Etapa 4: Como definir a interface crosscutting OutgoingMessage do aspecto Interaction?

A interface crosscutting OutgoingMessage define como interceptar as classes do conhecimento do agente interessadas em enviar mensagens internas ao ambiente. O aspecto `Interaction` define dois pointcuts abstratos `outgoingMsgFromRoles()` e `outgoingMsgFromPlans()` para alcançar esse comportamento necessário (linhas 79 e 86, respectivamente). Esses pointcuts especificam os join points nas classes do agente e do plano, respectivamente, quando uma mensagem deve ser enviada por um efetor do agente. Eles também precisam ser implementados pelos subaspectos Interaction (etapa 5). Cada um dos pointcuts possui um after advice associado que chama o método `sendMessage()` do aspecto Interaction (linhas 81-84 e 88-91, respectivamente).

```
79.   protected abstract pointcut outgoingMsgFromRoles(Agent agent);
80.
81.   after (Agent agent) returning (Message msg):
82.                            outgoingMsgFromRoles(agent) {
83.         sendMsg(msg, agent);
84.   }
85.
86.   protected abstract pointcut outgoingMsgFromPlans(Plan plan);
87.
88.   after (Plan plan) returning (Message msg):
```

```
89.                             outgoingMsgFromPlans(plan) {
90.         Agent agent = plan.getAgent();
91.         sendMsg(msg, agent);
92.   }
```

Etapa 5: Como definir um aspecto de interação concreto?

Os subaspectos Interaction definem implementações concretas do aspecto Interaction. Podemos especificar um subaspecto Interaction diferente para cada tipo de agente ou papel do agente definidos em nosso SMA. Os subaspectos devem implementar os pointcuts e métodos abstratos do aspecto Interaction, da seguinte maneira:

- pointcut `agentInstantiation()` – define um join point para inicializar os recursos de interação do agente ou papel;
- pointcuts `outgoingMessageFromRoles()` e `outgoingMessageFromPlans()` – definem, respectivamente, join points para enviar mensagens de papéis do agente e de planos do agente;
- método `initSensorsAndEffectors()` – cria e inicializa os sensores e os efetores usados por um tipo específico de agente ou papel.
- método `sendMsgEffector()` – define a lógica para decidir qual efetor do agente deve ser usado para enviar uma determinada mensagem.

No sistema EC, definimos os subaspectos Interaction `ChairInteraction` e `ReviewerInteraction` para especificar diferentes recursos de interação para os papéis do agente (chair e revisor). Apresentamos o código-fonte `ReviewerInteraction` a seguir.

```
1. public aspect ReviewerInteraction extends Interaction  {
2.    public pointcut agentInstantiation(Agent agent):
```

```
3.                this(agent) &&
4.                initialization(ResearcherUserAgent+.new(..));
5.
6.    protected pointcut outgoingMsgFromRoles(Agent agent):
7.                      target(agent) &&
8.                      execution(Message
Agent+.prepareMessage(..));
9.
10.   protected pointcut outgoingMsgFromPlans(Plan plan) :
11.       this(plan) &&
12.       execution(Message
ProposalReceptionPlan.prepareMessage(..));
13.
14.   public void sendMsgEffector(Message msg, Agent agent) {
15.         Hashtable effectors = agent.getEffectors();
16.         Effector agentEffector =
17.                     (Effector) effectors.get("Reviewer");
18.               agentEffector.sendMsg(msg);
19.   }
20.
21.   public void initSensorsAndEffectors(Agent agent) {
22.         Effector agentEffector = new BlackboardEffector();
23.
24.         Hashtable effectors = agent.getEffectors();
25.         effectors.put("Reviewer", agentEffector);
26.   }
27. }
```

O padrão Adaptação

Etapa 1: Como definir uma classe de adaptação abstrata?

O aspecto `Adaptation` é declarado como abstrato (linha 1) uma vez que precisa ser redefinido em diferentes contextos, ou seja, diferentes tipos de agente e papéis. Ele intercepta o método `receiveMsg()` introduzido na classe `Agent` pelo aspecto `Interaction` (consulte padrão Interação). Isso é necessário a fim de implementar a adaptação do conhecimento; pode ser necessário adaptar as crenças do agente (etapa 3) com base no conteúdo da mensagem.

Para implementar a adaptação do comportamento, o aspecto `Adaptation` intercepta três importantes join points: (i) as execuções de `setGoal()` na classe `Agent` para determinar um novo plano do agente que deve ser executado uma vez que foi definido um novo objetivo (etapa 4), (ii) as execuções de `executePlan()` nas subclasses `Plan` para retirar o plano da lista de planos a serem executados sempre que

concluir com sucesso ou (iii) para determinar um plano alternativo sempre que surgir uma exceção durante a execução do plano atual.

O aspecto `Adaptation` é implementado em AspectJ como um aspecto único porque ele afeta muitas classes no sistema. Nesse sentido, a instância `Agent` é passada como um parâmetro nos advices do aspecto `Adaptation` de forma que o código do advice possa determinar qual agente do sistema está para ser adaptado.

Etapa 2: Como resolver o conflito entre Autonomia e Adaptação?

Os aspectos Autonomy e Adaptation interceptam o mesmo join point: as execuções do método `receiveMsg()` na classe `Agent`. Os dois precisam interceptar a recepção de mensagens para respectivamente criar objetivos e adaptar o conhecimento do agente. Entretanto, a ordem de execução dos aspectos é importante em alguns casos. O aspecto Adaptation precisa ser executado antes do aspecto Autonomy. O construto declare precedence (linha 3) da linguagem AspectJ [12] foi usado a fim de implementar essa relação de dependência entre esses diferentes aspectos. Esse construto indica que a ordem dos aspectos relacionados é significativa.

```
1. public abstract aspect Adaptation {
2.
3.    declare precedence : Adaptation, Autonomy;
4.    ...
```

Etapa 3: Como definir a adaptação de crenças?

A adaptação de crenças do aspecto `Adaptation` é definida por um pointcut, um advice e um método. O pointcut `beliefAdaptation()` (linhas 5-7) intercepta o método `receiveMessage()` da classe `Agent`. Ele expõe como parâmetros as instâncias da mensagem e do agente (linha 5) e está associado a um after advice (linhas 9-11). Esse advice apenas chama o método `adaptBelief()` (linha 13-16).

O método `adaptBelief()` é decomposto em dois métodos: (i) `adaptGeneralBelief()` – implementa adaptações de crenças comuns a todos os papéis e tipos de agente (linhas 18-20); e (ii) `adaptSpecificBelief()` – um método abstrato (linhas 22-23) que precisa ser implementado pelos subaspectos Adaptation e que é responsável pela definição da adaptação de crenças específicas a diferentes papéis e tipos de agente.

```
5.    pointcut beliefAdaptation(Agent agent, Message msg):
6.          (args(msg) && this(agent)
7.            && execution(void Agent.receiveMsg(Message)));
8.
9.    after(Agent agent, Message msg) : beliefAdaptation(agent, msg)
{
10.         adaptBelief(agent, msg);
11.   }
12.
13.   public void adaptBelief(Agent agent, Message msg){
14.         adaptGeneralBelief(agent, msg);
15.         adaptSpecificBelief(agent, msg);
16.   }
17.
18.   public void adaptGeneralBelief(Agent agent, Message msg){
19.         ...
20.   }
21.
22.   abstract protected void adaptSpecificBelief
23.                                 (Agent agent, Message msg);
```

Etapa 4: Como definir a adaptação do comportamento?

O aspecto `Adaptation` define a adaptação do plano do agente por meio de dois pointcuts: (i) `planAdaptation()`(linhas 24-26) – responsável por interceptar o método `setGoal()` da classe `Agent` para determinar um plano do agente para alcançar um objetivo específico; e (ii) `exceptionalPlanAdaptation()` (linhas 53-54) – intercepta o método `executePlan()` das subclasses Plan para determinar a execução de um novo plano quando a execução de um plano falha.

O pointcut `planAdaptation()` associou um after advice que tenta encontrar um plano específico para alcançar o novo objetivo do agente (linhas 28-43). Os subaspectos Adaptation também podem definir planos específicos para alcançar objetivos implementando o método abstrato `findSpecificPlan()`. A lista de planos reativos do agente é verificada para permitir a seleção de um plano que pode alcançar o objetivo e satisfaz suas precondições. O plano selecionado é adicionado à lista de planos que deverão ser executados da classe `Agent` (linha 41).

O pointcut `exceptionalPlanAdaptation()` possui um "advice depois do levantamento" (linhas 56-59) associado a ele. Esse advice é executado sempre que o método `executePlan()` das subclasses Plan levanta uma exceção. Ele é responsável por descobrir outro plano para alcançar o objetivo que o plano atual não conseguiu alcançar.

```
24.   pointcut planAdaptation(Agent agent, Goal agentGoal):
25.               (args(agentGoal) && this(agent)
26.                 && execution(void setGoal(Goal)));
27.
28.   after(Agent agent, Goal agentGoal) returning():
29.         planAdaptation(agent, agentGoal) {
30.
31.         Hashtable agentPlans = new Hashtable();
32.
33.         // tries to find firstly an specific plan
34.         Plan plan = findSpecificPlan(agent, agentGoal);
35.         // tries to find an generic plan
36.         if (plan == null) {
37.               agentPlans = agent.getReactivePlanList();
38.               plan = findPlan(agent, agentGoal, agentPlans);
39.         }
40.         if (plan != null) {
41.               agent.addPerformPlan(plan);
42.         }
43.   }
44.
45.   public abstract Plan findSpecificPlan(Agent agent,
46.                                         Goal agentGoal);
47.
48.   public Plan findPlan(Agent agent,
49.                        Goal agentGoal, Hashtable agentPlans) {
```

```
50.         ...
51.   }
52.
53.   pointcut exceptionalPlanAdaptation():
54.                     call(public void Plan+.executePlan(..));
55.
56.   after() throwing(FailedPlanException exception) :
57.                             exceptionalPlanAdaptation() {
58.         ...
59.   }
60    ...
```

Etapa 5: Como definir um aspecto de adaptação concreto?

Os subaspectos Adaptation definem implementações concretas do aspecto `Adaptation`. Podemos especificar um subaspecto Adaptation diferente para cada tipo de agente ou papel do agente definidos no SMA. Os subaspectos devem implementar os dois métodos abstratos do aspecto `Adaptation`: (i) `adaptSpecificBelief()` e (ii) `findSpecificPlan()`.

O método `adaptSpecificBelief()` deve ser implementado para oferecer uma interpretação de uma mensagem específica de um tipo de agente ou papel. Esse método é chamado pelo aspecto `Adaptation` quando uma nova mensagem é recebida pelo agente. Ele é implementado para atualizar crenças específicas de tipos de agente e papéis do agente com base em mensagens recebidas do ambiente.

O método `findSpecificPlan()` é usado para incluir planos específicos definidos para tipos de agente ou papéis no processo de adaptação do plano. Ele deve ser implementado para obter a lista de planos específicos de um tipo de agente (ou papel) e deve passá-la para o método `findPlan()` herdado de `Adaptation aspect`. O código-fonte do aspecto `ChairAdaptation` do sistema EC é apresentado a seguir.

```
1. public aspect ChairAdaptation extends Adaptation {
2.
3.    protected pointcut agentInstantiation(Agent agent):
4.          this(agent) &&
5.          initialization(ResearcherUserAgent+.new(..));
6.
7.    protected void adaptSpecificBelief(Agent agent, Message msg){

8.          String performative = msg.getPerformative();
9.          Hashtable content = (Hashtable) msg.getContent();
10.
11.         String service=null;
12.         String performativeService=null;
13.
14.         if (performative.equals(FIPATypes.REQUEST)) {
15.               service = (String)
16.
content.get(MessageConstants.SERVICE);
17.               performativeService = performative + service;
18.
19.               if (performativeService.equals(FIPATypes.REQUEST+
20.
      MessageConstants.SERVICE_DISTRIBUTE_PAPERS)){
21.
      updateInformationAboutPaperDistribution(agent,
22.
msg);
23.               }
24.         }
25.   }
26.   public Plan findSpecificPlan(Agent agent, Goal agentGoal) {
27.         ResearcherUserAgent researcher =
28.                                    (ResearcherUserAgent)agent;
29.         Hashtable plans = researcher.getReactiveChairPlanList();

30.         Plan plan = null;
31.         plan = findPlan(agent, agentGoal, plans);
32.         return plan;
33.   }
34.   private void updateInformationAboutPaperDistribution(Agent
agent,
35.                                                      Message msg){
36.         ...
37.   }
38. }
```

O padrão Autonomia

Etapa 1: Como definir uma classe de autonomia abstrata?

O aspecto `Autonomy` é declarado como abstrato uma vez que precisa ser redefinido em diferentes contextos, ou seja, diferentes tipos de agente e papéis. O

propósito básico do aspecto Autonomy é instanciar a gerenciar objetivos reativos. Ele oferece esse serviço ao implementar o pointcut `reactiveGoalsInstantiation()` (linhas 3-5). Esse pointcut intercepta o método `receiveMsg()` introduzido na classe `Agent` pelo aspecto `Interaction`. Depois desse desvio no fluxo de controle do programa, ele verifica se a mensagem externa demanda a instanciação de algum objetivo reativo (linha 9). Nesse caso, ele executa um método para verificar a instanciação possível de um objetivo reativo geral – comum a todos os agentes (linha 15) – ou um objetivo reativo específico (linha 16) – que deve ser implementado pelos subaspectos Autonomy.


```
1. public abstract aspect Autonomy {
2.
3.    pointcut reactiveGoalsInstantiation (Agent agent, Message
msg):
4.          (args(msg) && this(agent) &&
5.           execution(void Agent.receiveMsg(Message)));
6.
7.    after(Agent agent, Message msg):
8.                  reactiveGoalsInstantiantion(agent, msg) {
9.          boolean decision = makeDecision(agent, msg);
10.         if (decision){
11.               instantiateReactiveGoal(agent, msg);
12.         }
13.   }
14.   public void instantiateReactiveGoal(Agent agent, Message msg){
15.         instantiateGeneralReactiveGoal(agent, msg);
16.         instantiateSpecificReactiveGoal(agent, msg);
17.   }
18.   public void instantiateGeneralReactiveGoal(Agent agent,
19.                                              Message msg){
20.         ...
21.   }
22.   public abstract void instantiateSpecificReactiveGoal(Agent
agent,
23.                                                    Message
msg);
```

As implementações sofisticadas do aspecto Autonomy definem três tipos de autonomia: (i) autonomia proativa; (ii) autonomia reativa; e (iii) autonomia de

execução. Descrevemos a seguir (etapas 2, 3 e 4) como implementar esses elementos. Os subaspectos Autonomy refinam esses tipos de autonomia para um papel ou tipo de agente específico (etapa 5).

Etapa 2: Como definir a autonomia proativa?

A autonomia proativa deve especificar vários pointcuts nas classes do conhecimento do agente que representam eventos específicos de interesse (linhas 25-27). Ela se associa com esses pointcuts, um after advice (linhas 29-35) que chama o método `makeDecision()` (linhas 31). Esse método é responsável por determinar se um objetivo proativo deve ser instanciado devido à ocorrência de eventos. Os objetivos proativos gerais devem ser instanciados no aspecto `Autonomy` e objetivos proativos específicos devem ser inicializados nos subaspectos Autonomy (etapa 5).

```
24.
25.   pointcut proactiveDecisionMaking(Agent agent, Event event):
26.         (args(stimulus) && this(agent) &&
27.          execution(void Agent.setEvent(Event+)));
28.
29.   after(Agent agent, Event event):
30.               proactiveDecisionMaking(agent, event) {
31.         boolean decision = makeDecision(agent, event);
32.         if (decision){
33.               instantiateProactiveGoal(agent, event);
34.         }
35.   }
36.   public void instantiateProactiveGoal(Agent agent, Event
event){
37.         instantiateGeneralProactiveGoal(agent, event);
38.         instantiateSpecificProactiveGoal(agent, event);
39.   }
40.   public void instantiateGeneralProactiveGoal(Agent agent,
41.                                                Event event){
42.         ...
43.   }
44.   public abstract void
45.         instantiateSpecificProactiveGoal(Agent agent, Event
event);
46.   ...
```

A implementação da autonomia proativa define duas estruturas de dados (tabelas de hash) para associar: (i) eventos internos (mudança de crença, início ou fim da execução de um plano) com objetivos proativos; e (ii) objetivos proativos com uma lista de planos que permite alcançar cada objetivo. Essas estruturas de dados são usadas para oferecer suporte a instanciações de objetivos proativos durante a ocorrência de um evento específico. As estruturas similares são usadas na implementação da autonomia de decisão (etapa 3).

Etapa 3: Como definir a autonomia de definição?

A autonomia de decisão age, quando apropriado, antes das instanciações de objetivos proativos e reativos. Dessa forma, a implementação da autonomia de decisão define o método `makeDecision()` para verificar antes da instanciação desses tipos de objetivo se for necessário instanciar qualquer objetivo de decisão geral ou específico (linhas 50-53). Nesse caso, um plano de decisão específico ou geral é executado para decidir se é necessário ao agente alcançar o objetivo (linha 54-67).

O método `makeDecision()` é chamado nos advices associados aos pointcuts relacionados a uma instanciação de objetivo proativo e de objetivo reativo.

```
47.  public boolean makeDecision(Agent agent, Stimulus stimulus) {
48.         boolean decision = true;
49.
50.         Vector generalDecisionGoals =
51.               instantiateGeneralDecisionGoal(agent, stimulus);
52.         Vector specificDecisionGoals =
53.               instantiateSpecificDecisionGoal(agent,stimulus);
54.         Enumeration generalDecisions =
55.               generalDecisionGoals.elements();
56.         Enumeration specificDecisions =
57.               specificDecisionGoals.elements();
58.
59.         // performing the general decisions
60.         while (generalDecisions.hasMoreElements()&& decision){
61.               ...
62.         }
63.         // performing the specific decisions
```

```
64.         while (specificDecisions.hasMoreElements() && decision)
{
65.               ...
66.         }
67.         return decision;
68.   }
```

Etapa 4: Como definir a autonomia de execução?

A autonomia de execução é implementada no aspecto Autonomy ao definir um objeto ativo [153]. Uma estratégia de concorrência (como, pool de thread ou thread por solicitação) e a lista de planos do agente que devem ser executados são passadas como argumentos para o objeto ativo (linhas 78-80). A primeira é usada para definir uma política de criação de threads. A segunda é monitorada pelo objeto ativo de forma que quando um novo plano é inserido na lista, ele pode executado em uma thread separada.

```
69.   ...
70.   private ActiveObject autonomyActiveObject;
71.   private int maxNumberOfThreadsPerAgent = 5;
72.
73.   public void initThread(Agent agent) {
74.         // Instantiate the ConcurrencyStrategy
75.         ConcurrencyStrategy concurrencyStrategy =
76.             new ThreadPoolStrategy(1,
maxNumberOfThreadsPerAgent);
77.
78.         autonomyActiveObject = new ActiveObject(
79.                                  agent.getToPerformPlans(),
80.                                  concurrencyStrategy);
81.         autonomyActiveObject.start();
82.   }
83.   ...
```

Etapa 5: Como definir um aspecto de autonomia concreto?

Os subaspectos Autonomy definem implementações concretas do aspecto Autonomy. Podemos especificar um subaspecto Autonomy diferente para cada tipo de agente ou papel do agente definidos em nosso SMA. Os subaspectos devem

implementar os pointcuts e métodos abstratos do aspecto `Autonomy`, da seguinte maneira:

- pointcut `agentInstantiation()` – define um join point para inicializar os recursos de autonomia do agente;
- os pointcuts `initSpecificGoals()` e `initSpecificPlans()` – permitem especificar, respectivamente, estruturas de dados diferentes para associar estímulos (evento, mensagens externas) com objetivos e associar objetivos a planos;
- método `instantiateSpecificReactiveGoal()` – define o comportamento para determinar se é necessário instanciar um objetivo reativo específico devido a uma mensagem externa;
- método `instantiateSpecificProactiveGoal()` – define o comportamento para determinar se é necessário instanciar um objetivo proativo específico devido a uma ocorrência de evento;
- método `instantiateSpecificDecisionGoal()` – define o comportamento para determinar se é necessário instanciar um objetivo de decisão específico devido a uma ocorrência de um estímulo.

O padrão Papel

Etapa 1: Como definir um aspecto de papéis?

O aspecto de papéis deve introduzir atributos e métodos em um tipo de agente (subclasses ou classe `Agent`). Esses elementos definem, respectivamente, crenças específicas e comportamento de um papel colaborativo. Além disso, classes do objetivo e plano específico devem ser definidos para o papel. Os planos definidos

para um papel podem manipular os atributos (crenças) a chamar métodos (comportamento) introduzidos pelo aspecto de papéis. As classes Goal especificadas para um papel são usadas pelo subaspecto Autonomy definido para o papel apropriado.

No sistema EC, definimos os aspectos de papéis `Chair` e `Reviewer`. O aspecto `Chair`, por exemplo, introduz na classe `ResearchUserAgent` (uma subclasse `Agent`), vários atributos e métodos para tornar possível que instâncias dessa classe assumam o papel de colaboração de chair. Alguns exemplos de atributos introduzidos pelo aspecto `Chair` são: um plano de distribuição de trabalhos para o comitê do programa (linha 3), uma lista de trabalhos enviados (linha 4), uma lista de revisores de trabalhos (linha 5) e a data limite para enviar trabalhos (linhas 6-7).

```
1. public aspect Chair {
2.    ...
3.    private DistributionPlan ResearcherUserAgent.distributionPlan;
4.    private List ResearcherUserAgent.papersList;
5.    private List ResearcherUserAgent.reviewerList;
6.    private GregorianCalendar
7.                      ResearcherUserAgent.paperSubmissionDeadline;
8.    ...
```

Também foram definidas no sistema EC as classes do objetivo e plano específico para os aspectos de papéis `Chair` e `Reviewer`. Por exemplo, um objeto `ResearchUserAgent` que exerce o papel de chair deve executar um plano de distribuição de trabalho, que é responsável por garantir a alocação de trabalhos enviados aos revisores. Assim, as classes `PaperDistributionGoal` e `PaperDistributionPlan` foram definidas para esse propósito.

Quando um agente exerce um papel, ele normalmente requer a definição de novos recursos de interação, autonomia e adaptação. Nesse caso, precisamos definir

novos subaspectos dos aspectos `Interaction`, `Autonomy` e `Adaptation`. Os subaspectos Interaction devem definir sensores e efetores específicos usados pelo papel. Os subaspectos Autonomy devem inicializar e instanciar planos e objetivos específicos usados pelo papel. Finalmente, os subaspectos Adaptation devem definir estratégias específicas para atualizar crenças de papel e selecionar planos para alcançar objetivos do papel. As seções de implementação anteriores dos padrões Interação, Autonomia e Adaptação apresentam as diretivas para definir esses subaspectos.

A implementação do padrão Papel em AspectJ limita a capacidade de acoplar e desacoplar aspectos de papéis de forma dinâmica dos objetos do agente, porque a versão atual dessa linguagem (1.1.4) não oferece suporte ao processo de combinação dinâmico. Como essa característica é importante para a implementação dos papéis do agente, as tecnologias dos processos de combinação dinâmicos (por exemplo [13, 76, 196]) podem ser usadas para facilitar a reconfiguração dinâmica de um SMA. Nesse caso, o aspecto `Adaptation` deve ser responsável por associar dinamicamente aspectos de papéis a objetos do agente.

O padrão Aprendizagem

Etapa 1: Como definir um aspecto Learning?

O aspecto `Learning` deve definir o protocolo de aprendizagem, que envolve observar e interceptar objetos de conhecimento específicos e, depois disso, chama algoritmos de aprendizagem específicos. Em AspectJ, usamos pointcuts para definir quais join-points na execução de objetos, o aspecto `Learning` pode ter interesse.

Esses pointcuts devem expor como parâmetros, informações (instâncias de objeto) necessárias para serem usadas nos algoritmos de aprendizagem. Os advices associados a esses pointcuts chamam métodos do aspecto `Learning` ou algum componente Learning e, se necessário, eles passam as informações reunidas nos pointcuts como argumentos.

No sistema EC, implementamos o aspecto `Learning` nos papéis do agente revisor e chair. Um agente que exerça o papel de chair aprende novos interesses de pesquisa de um revisor específico com base na resposta da proposta do revisor do trabalho. Um agente que exerce o papel de revisor aprende novos interesses de pesquisa de um usuário com base em feedbacks. Como o aprendizado de chair e do revisor tem aspectos comuns e estão inter-relacionados, implementamos seu comportamento em uma hierarquia de aspectos, composta pelo aspecto `Learning` e subaspectos `ChairLearning` e `ReviewerLearning`.

O aspecto `Learning` define o pointcut abstrato `interestDegreeLearning()` (linhas 23-25) responsável por declarar join-points na execução do objeto em que os algoritmos de aprendizagem podiam ser chamados. Ele também especifica os métodos: (i) `learnAgentPreference()` (linhas 27-33) – que implementa um algoritmo de aprendizagem com base na técnica Temporal Difference (TD-Learning); e (ii) `updatePreferences()` (linhas 34-37) – que garante a atualização dos interesses de pesquisa do usuário, depois da execução do algoritmo de aprendizagem. Ademais, o aspecto `Learning` introduz os atributos `proposalEvaluation` e `currentPaperInterest` (linhas 3-6) na classe `RevisionProposal`, a fim de tornar possível que o papel do agente chair possa aprender com base na avaliação do revisor.

```
1. public aspect Learning {
2.    ...
3.    private Hashtable RevisionProposal.proposalEvaluation =
4.                                                 new Hashtable();
5.
6.    private int RevisionProposal.currentPaperInterest = 0;
7.
8.    public Hashtable RevisionProposal.getEvaluation() {
9.          return proposalEvaluation;
10.   }
11.
12.   public void RevisionProposal.setEvaluation(Hashtable
evaluation){
13.         this.proposalEvaluation = evaluation;
14.   }
15.
16.   public int RevisionProposal.getPaperInterest(){
17.         return currentPaperInterest;
18.   }
19.
20.   public void RevisionProposal.setPaperInterest(int interest) {
21.         this.currentPaperInterest = interest;
22.   }
23.   protected abstract pointcut interestDegreeLearning(
24.                                RevisionProposal proposal,
25.                                Plan plan);
26.
27.   public Hashtable learnAgentPreference(Hashtable
currentInterests,
28.                           Vector my_keywords, boolean
newDecision,
29.                           int currentPaperInterestDegree) {
30
31.         // Implements a TD-Learning algorithm
32.         ...
33.   }
34.   public void updatePreferences(Hashtable currentInterests,
35.                                Hashtable newPreferences) {
36.         ...
37.   }
38.   ...
39. }
```

`ChairLearning` implementa o pointcut `interestDegreeLearning()` ao interceptar o método `verifyReviewerResponse()` da classe `ProposalJudgementReceptionPlan` (linhas 31-36). Esse pointcut está associado a um after advice (linhas 38-64), que é responsável por avaliar a proposta de revisão do

trabalho devolvida pelos revisores (chamando o método `learnAgentPreference()`) e atualizar os interesses de pesquisa (chamando o método `updatePreferences()`), usando as informações (atributos `proposalEvaluation` e `currentPaperInterest`) introduzidas na classe `RevisionProposal`. `ChairLearning` ainda especifica o atributo `reviewers` (linha 4) que mantém o conhecimento sobre os interesses de pesquisa dos revisores, usado para oferecer suporte à aprendizagem da chair. O atributo `reviewer` é inicializado, pelo pointcut `learningInitialization()` (linhas 10-15) e seus respectivos advice (linhas 17-30), antes da execução do método `sendPapersToReviewer()` da classe `PaperDistributionPlan`.


```
1. public aspect ChairLearning extends Learning {
2.    ...
3.    // //(reviewer name, table of research interests)
4.    public Hashtable reviewers = new Hashtable();
5.
6.    protected pointcut agentInstantiation(Agent agent):
7.                this(agent) &&
8.                initialization(ResearcherUserAgent+.new(..));
9.
10.   protected pointcut learningInitialization(Agent agent,
11.                           Reviewer reviewer, List papers):
12.         args (agent, reviewer, papers) &&
13.         call(public void
14.               PaperDistributionPlan.sendPapersToReviewer(
15.                                     Agent, Reviewer, List));
16.
17.   before (Agent agent, Reviewer reviewer, List papers):
18.          learningInitialization(agent, reviewer, papers) {
19.         String reviewerName = reviewer.getName();
20.         Hashtable reviewer_interests =
21.                     (Hashtable) reviewers.get(reviewerName);
22.
23.         if (reviewer_interests == null) {
24.               reviewer_interests = new Hashtable();
25.               reviewers.put(reviewerName, reviewer_interests);
26.         }
27.
28.         // Initialize the research interest of the reviewer
29.         ...
30.   }
31.   protected pointcut interestDegreeLearning(
32.                           RevisionProposal proposal, Plan plan):
(
33.      this(plan) && args(proposal) &&
```

```
34.         execution(void
35.
ProposalJudgementReceptionPlan.verifyReviewerResponse(
36.
RevisionProposal)));
37.
38.   after (RevisionProposal proposal, Plan plan):
                    interestDegreeLearning(proposal, plan) {
40.
41.         boolean acceptedProposal = proposal.isAccepted();

42.         Paper paper = proposal.getPaper();
43.         ResearchArea area = paper.getResearchArea();
44.         Vector paperKeywords = area.getResearchKeywords();
45.
46.         Hashtable reviewerEvaluation = proposal.getEvaluation();
47.         int reviewerInterest = proposal.getPaperInterest();
48.
49.         Reviewer reviewer = proposal.getReviewer();
50.         String reviewerName = reviewer.getName();
51.
52.         //getting the reviewer' current interests
53.         Hashtable reviewerPreferences = (Hashtable)
54.                                 reviewers.get(reviewerName);

55.
56.         //learning the new user interests
57.         Hashtable newPreferences =
58.                     learnAgentPreference(reviewerPreferences,
59.                                          paperKeywords,
60.                                          acceptedProposal,
61.                                          reviewerInterest);

62.         //update my preferences
63          updatePreferences(reviewerPreferences,newPreferences);
64.   }
65. }
```

O subaspecto `ReviewerLearning` define o pointcut `interestDegreeLearning()` ao interceptar o método `judgeProposal()` da classe `ProposalReceptionPlan`. Esse pointcut está associado a um after advice, que chama o método `learnAgentPreference()` passando as informações sobre a proposta de revisão de trabalho, causando a atualização dos interesses de pesquisa do revisor com base em sua avaliação.

O padrão Mobilidade

Descrevemos a seguir as partes do código que são relevantes para entender como implementar o padrão Mobilidade usando a linguagem AspectJ e o framework JADE.

Etapa 1: Como definir um aspecto Mobility?

O aspecto `Mobility` é declarado como abstrato (linha 1) uma vez que precisa ser redefinido em diferentes contextos, ou seja, diferentes tipos de agente e papéis. Ele introduz o atributo booleano `agentOut` em classe `Agent` para indicar que o agente se moveu para outro ambiente (linha 8). Ele declara dois atributos: (i) `location` (linha 6) – para armazenar o local atual do agente; e (ii) uma componente de mobilidade – para fornecer serviços para que o agente se mova para outro ambiente (no sistema EC, foi usado um componente de mobilidade implementado usando o framework JADE. Está detalhado a seguir).

Vários pointcuts e métodos abstratos são declarados no aspecto `Mobility` a ser implementado por seus subaspectos, como:

pointcut `moveAgent()` (linha 37) – usado para definir join-points na execução do agente, onde ele poderia ser movido para outro ambiente;

pointcut `afterMove()` (linha 48) – usado para definir o join-point específico que identifica que um agente foi movido;

método `init()` (linha 17) – usado para inicializar os recursos de mobilidade do agente em uma determinada plataforma;

método `move()` (linha 19) – usado para definir as ações que devem ser executadas para mover um agente para outro ambiente;

método `initializeAgentInNewEnviroment()` (linhas 53-54) – usado para definir a inicialização do agente depois do deslocamento;

método `checkMobilityNeed()` (linhas 56-57) – identifica a necessidade de mobilidade do agente.

No sistema EC, o framework JADE foi usado para oferecer recursos de mobilidade aos agentes. Definimos um componente de mobilidade (`JADECommunicationModule` class) ao dividir em subclasses a classe `jade.core.Agent` (linha 5). Dessa forma, o componente de mobilidade pode fornecer vários métodos de mobilidade na plataforma JADE (como `doMove()`, `beforeMove()`, `afterMove()`). Ao usar JADE, também foi necessário garantir que as classes de conhecimento (`Agent`, `Plan`, `Goal`) implementem a interface `Serializable`, para possibilitar a migração do objeto através da rede (linhas 2-3). A construção “declare parents” de AspectJ foi usada para esse propósito.

```
1. public abstract aspect Mobility {
2.    declare parents:
3.          Agent || AbstractPlan || Goal implements Serializable;
4.
5.    protected JADECommunicationModule mobilityModule;
6.    private Object location;
7.
8.    private boolean Agent.agentOut;
9.
10.   protected abstract pointcut agentInstantiation(Agent agent);
11.
12.   after(Agent agent) : agentInstantiation(agent) {
13.         agent.setAgentOutFalse();
14.         init(agent);
15.   }
16.
17.   public abstract void init(Agent agent);
18.
19.   public abstract void move(Agent agent);
20.
21.   public Message Agent.prepareDepartureNotification() {
22.         ...
23.   }
```

```
24.   public Message Agent.prepareArrivalNotification() {
25.         ...
26.   }
27.   public boolean Agent.isAgentOut() {
28.         return this.agentOut;
29.   }
30.   public void Agent.setAgentOutTrue() {
31.         this.agentOut = true;
32.   }
33.   public void Agent.setAgentOutFalse() {
34.         this.agentOut = false;
35.   }
36.   ...
37.   protected abstract pointcut moveAgent(Plan plan);
38.
39.   after(Plan plan) returning (Object result): moveAgent(plan) {
40.         Agent agent = plan.getAgent();
41.         boolean moveAgent = checkMobilityNeed(agent, result);
42.         if (moveAgent && (!agent.isAgentOut())) {
43.               agent.prepareDepartureNotification();
44.               move(agent);
45.         }
46.   }
47.
48.   protected abstract pointcut afterMove(Object mobileAgent);
49.
50.   after(Object mobileAgent) : afterMove(mobileAgent) {
51.         initializeAgentInNewEnvironment(mobileAgent);
52.   }
53.   public abstract void initializeAgentInNewEnvironment(
54.                                       Object mobileAgent);
55.
56.   protected abstract boolean checkMobilityNeed(Agent agent,
57.                                                Object result);
58.   public abstract void Agent.move(Agent agent);
59.
60.   ...
61. }
```

Etapa 2: Como definir um aspecto de mobilidade concreto?

Os subaspectos Mobility definem implementações concretas do aspecto `Mobility`. Podemos especificar um subaspecto Mobility diferente para cada tipo de agente ou papel do agente definidos em nosso SMA. Os subaspectos devem implementar os pointcuts e métodos abstratos do aspecto `Mobility`. Por exemplo, no sistema EC esses elementos foram implementados da seguinte forma:

- pointcut `moveAgent()` – define a interpretação do método `searchInformation()` da classe `InformationSearchPlan`, como um possível ponto de execução onde o agente deve ser movido;

- pointcut `afterMove()` – intercepta o método `afterMove()` da classe `JADECommunicationModule`;

- método `init()` – define a inicialização e a configuração dos recursos de mobilidade do agente na plataforma JADE;

- método `move()` – chama o método `doMove()` da classe `JADECommunicationModule`;

- método `initializeAgentInNewEnviroment()` – define ações que devem ser executadas depois que o agente se moveu para outro ambiente (como, notifica outros agente, instala sensores e efetores no novo ambiente, reinicializa os planos de interrupção);

- método `checkMobilityNeed()` – verifica se a execução do plano de busca de informações obteve algum resultado.

Apêndice II
Regras de transformação: De AspectJ a Hyper/J

Este apêndice descreve as regras para transformar programas AspectJ em programas Hyper/J. Primeiro, os recursos Hyper/J são apresentados. Segundo as regras de transformação são descritas. Terceiro, as principais descobertas do estudo de mapeamento (Seção 7.3) são discutidas.

Hyper/J

Hyper/J [235] é uma ferramenta desenvolvida no Centro de Pesquisa Watson da IBM para oferecer suporte a Hyperspaces [235], uma evolução da programação orientada a sujeitos [116]. Hyper/J oferece suporte à identificação, encapsulação e integração de concerns de programas Java padrão, sem exigir extensões de linguagem especiais, convenções de codificação ou empacotamento.

Em Hyperspaces, um *hyperspace* é definido como um conjunto de unidades de softwares que compõem um sistema de software, organizado em uma matriz multidimensional, onde cada eixo representa uma *dimensão do concern*, cada ponto em um eixo representa um *concern nessa dimensão*, e as coordenadas da unidade de um software indicam todos os concerns que afeta. Cada dimensão do concern pode ser vista como uma partição do conjunto de unidades: uma decomposição de software particular. Os principais elementos de Hyperspaces que têm o suporte de Hyper/J são:

- *Hyperslices*: Um hyperslice é um conjunto de unidades (pacotes, classes, métodos, campos etc.) que incorporam um concern; hyperslices são declarativamente completas, ou seja, precisam declarar tudo aquilo a que se referem. Um hyperslice generaliza o conceito de *sujeito* da programação orientada a sujeitos.
- *Unidades correspondentes*: As unidades correspondentes são unidades (pacotes, classes, métodos, campos etc.) que pertencem a diferentes hyperslices e que se correspondem entre si de acordo com alguns critérios. As unidades correspondentes devem ser compostas (com relacionamentos de integração por sobreposição ou merge) para formar uma nova unidade de software que contém algumas ou todas as funcionalidades das unidades originais.
- *Estratégia de composição*: A especificação dos relacionamentos de integração em Hyper/J segue uma abordagem em que os desenvolvedores primeiro especificam uma estratégia geral para identificar unidades correspondentes em hyperslices e, em seguida, definir as exceções, ou especializações, dessa estratégia para esses casos em que a estratégia não se aplica. Hyper/J oferece suporte a três estratégias gerais: *mergeByName*, *nonCorrespondingMerge* e *overrideByName*.
- *Precedência*: Hyper/J oferece regras para estabelecer a precedência relativa entre hyperslices de composição. O relacionamento de *ordem* indica que a ordem das unidades relacionadas é importante e descreve as restrições de ordem.

Antes de usar a ferramenta Hyper/J em um conjunto de classes java, o desenvolvedor deve fornecer três resultados: um arquivo hyperspace, um ou mais arquivos de mapeamento de concern e um arquivo hipermódulo. O *arquivo hyperspace* é semelhante a uma descrição de projeto. Ele lista todos os arquivos da classe Java com a qual um desenvolvedor está trabalhando e ao qual ele deseja aplicar Hyper/J. O(s) *arquivo(s) de mapeamento de concern* descreve(m) como as diferentes partes dos arquivos da classe Java cuidam de diferentes concerns em um hyperspace. Por exemplo, um desenvolvedor pode declarar que qualquer método chamado exibir-algo trata do concern Exibir. O *arquivo hipermódulo* descreve quais hyperslices devem ser integrados e como a integração deve prosseguir, incluindo a estratégia de composição geral, condições *equate*, condições *order* etc. Uma vez escritos os arquivos descritos acima, a ferramenta do compositor Hyper/J pode ser aplicada para produzir um novo conjunto de unidades integradas – um novo hyperslice composto.

Regras de transformação

Um conjunto de regras de transformação foi aplicado seguindo as etapas descritas a seguir:

1. Separar arquivos com código Java padrão de arquivos com código AspectJ[12].
2. Criar um novo arquivo .hs (arquivo hyperspace), declarando os nomes das classes que estarão no hyperspace.
3. Criar um novo arquivo .hm (arquivo hipermódulo).
4. Para cada aspecto A_i, $i = 1, n$ e $j \neq i$ com cabeçalho

   aspecto A_i [domina A_j] de eachobject(instanceof(CName))

   a) Criar um pacote chamado A_i

   b) Criar um arquivo chamado A_i/concerns.cm contendo:

      pacote A_i: A_j.Kernel
5. Declarar no arquivo .hm

   a) O hyperslice "base"

   b) Os hyperslices A_i.Kernel, para cada aspecto A_i, $i = 1, n$ e $j \neq i$, contendo:

      aspecto A_i [domina A_j] de eachobject(instanceof(CName))

   onde, se A_i domina A_j, então A_i.Kernel é declarado depois de A_j.Kernel[13].

   c) A estratégia de composição geral: mergeByName
6. Para cada aspecto A_i, $i = 1, n$ e $j \neq i$ de forma que

   aspecto A_i [domina A_j] de eachobject(instanceof(CName))

[12] Os arquivos originais com Java padrão não foram reescritos.
[13] Para sobreposição por after advices.

a) Criar um pacote chamado A_i/Cname.java e declarar a classe Cname nele

b) Copiar introduções do aspecto A_i na classe Cname

c) Para cada pointcut que contém executions(T methodName):

   i. Definir novos métodos methodName com corpo B definido no advice

   ii. Se o advice definido no pointcut for before

   Declarar no arquivo .hm

   order action Ai.Kernel.CName.methodName
   before action Agent.Kernel.CName.methodName;

   iii. Se T ≠ vazio, criar uma função de resumo e declarar set summary function no arquivo .hm.

Descobertas

A conclusão foi que a arquitetura de agente proposta (Capítulo 4) e os padrões de projeto (Capítulo 5) também podem ser implementados em Hyper/J. As regras de transformação (Apêndice II) oferecem suporte a um mapeamento direto das soluções de projeto orientadas a aspectos para soluções baseadas no modelo de programação de Hyper/J. As duas ferramentas ofereciam suporte à separação e composição de concerns de agência conforme descrito pelas diretrizes arquiteturais propostas e a linguagem de padrões. Detectamos algumas similaridades entre os elementos de programação (Tabela 2). Além disso, temos estas conclusões [14]:

[14] É importante ressaltar que as conclusões deste estudo limitaram-se às versões específicas das ferramentas usadas no nosso estudo e os recursos usados na implementação do Portalware.

- A dicotomia base-aspecto de AspectJ, com dependência explícita de alguma "base" cujo vocabulário é compartilhado entre os "aspectos", pode ser considerada uma disciplina que pode simplificar o desenvolvimento de soluções MDSoC, em especial as regras de composição. Skipper [224] compara a programação orientada a sujeitos e orientada a aspectos e chega a uma conclusão similar.
- Lai, Murphy e Walker [150] descrevem um experimento com Hyper/J e discutem alguma reestruturação de código para permitir capturar e compor concerns. Esse experimento compartilhou os benefícios e limitações relatados apresentados em [150], apesar de não termos reestruturado classes;e sim reescrevermos aspectos em uma ou mais classes. Durante o processo de transformação, os aspectos de AspectJ e os pointcuts chamados são bons mecanismos para melhorar a legibilidade do código crosscutting. O processo de transformação foi simplificado para reestruturar cada definição do aspecto em várias classes, reescrevendo o código de advice em métodos ordinários e usando completamente a estratégia de composição *mergeByName* em Hyper/J. No entanto, a solução pode perder alguns dos benefícios relatados proporcionados pelos aspectos e pelos pointcuts [6].
- No entanto, quando as condições iniciais relatadas na Seção 7.3.1 não são verdadeiras, a transformação pode requerer muito trabalho adicional e possivelmente algum tipo de refatoração em alguns métodos dentro da definição do aspecto. Uma fonte de problema da tradução para Hyper/J é algo que normalmente acontece em programas AspectJ: a adição de algo à base via

inter-type declarations e um advice subseqüente definido nela, dentro do mesmo aspecto. Como não podemos ter dois métodos com a mesma assinatura em uma classe, isso exigiria uma regra de reescritura mais complexa, com diferentes convenções de nomenclatura para o advice e a inclusão da outra condição *equate* em um arquivo hipermódulo.

- O experimento também permitiu a avaliação de como os mecanismos de composição de cada ferramenta funcionam na prática. Em AspectJ, mesmo com a força expressiva de wildcards em pointcuts, isso pode acabar sendo necessário para modificar invasivamente a definição de um aspecto, porque a composição entre aspectos e classes são *hardwired* no cabeçalho do aspecto. Além disso, o uso de *dominações* estabelece um relacionamento rígido entre aspectos e, conseqüentemente, entre todos os pointcuts. Seria interessante se o projetista pudesse especificar exceções à ordem de composição para alguns pointcuts. Hyper/J oferece uma solução mais elegante e flexível, ao requerer a definição da estratégia de composição explicitamente fora da definição de hyperslice e permitindo a especificação de exceções à estratégia de composição e à ordem de composição entre as unidades (também no nível de operações). Essa abordagem aumenta o potencial para a reconfiguração e modificação não-invasiva. Finalmente, vale a pena mencionar que um requisito importante declarado para o *Portalware* – o acoplamento de diferentes aspectos para instâncias distintas de agentes – não foi cumprido devido às limitações atuais dos mecanismos de composição das duas ferramentas.

- Acreditamos que se tivéssemos começado esse experimento com hyperslices em vez de aspectos, o processo de transformação traria algumas dificuldades, em parte porque: (i) os mecanismos de composição oferecidos por Hyper/J são mais poderosos e flexíveis do que aqueles fornecidos por AspectJ e (ii) teríamos mais trabalho para considerar os hyperslices sob a perspectiva da dicotomia base-aspecto.