

**Extensible MultiModal Environment Toolkit (EMMET):  
A Toolkit for Prototyping and Remotely Testing Speech and Gesture  
Based Multimodal Interfaces**

by

Christopher A. Robbins

A dissertation in partial fulfillment  
of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September, 2005

---

Dr. Kenneth Perlin

©Christopher A.Robbins  
All Rights Reserved, 2005

# Dedication

To my wife Julie, without whom none of this would be possible.

# Acknowledgments

I am profoundly grateful to Dr. Kenneth Perlin of New York University, who has served as my doctoral thesis advisor. I feel exceedingly privileged to have had the opportunity to learn from and interact with such a brilliant paragon, and I feel honored to have done so. It was Dr. Perlin's fascinating research that first attracted me to New York University, and it was Dr. Perlin's guidance, support, expertise, creativity, and open-mindedness that facilitated my abilities to rise to the many challenges inherent in the rigors of the doctoral program. Dr. Perlin has been my instructor both in and out of the classroom, and what I have learned from him transcends academic knowledge. The extent of my admiration, respect, and appreciation for Dr. Perlin truly defies the limits of the English language.

Within New York University, I have been exceptionally fortunate to have been surrounded by gifted minds and accomplished people. I would like to extend my most sincere gratitude to Dr. Denis Zorin, Director of Graduate Studies in the Department of Computer Science. Dr. Zorin not only accepted me into the doctoral program, thereby providing me with the incredible opportunity to be a part of a cutting-edge and exciting world of learning, but he also provided comprehensive departmental leadership throughout my experience at the University. Additionally, I am grateful to Dr. Margaret Wright, Chair of the Department of Computer Science. Dr. Wright's leadership has helped maintain the highest standards of excellence in the department of which I am so proud to have been a part.

Further, I would like to thank Dr. Alan Gottlieb and Dr. Zvi Kedem, under both of whom I had the opportunity to serve as a teaching assistant. I learned a great deal from both of these professors, and I am grateful to each of them for their support,

which continued long after my teaching assistantships were completed. I thank Dr. Christoph Bregler, who generously gave of his time and energy in my thesis proposal committee and my doctoral dissertation committee, and Dr. Dennis Shasha, who participated in my doctoral defense.

I was uniquely privileged to work with two additional professors from outside of NYU's Department of Computer Science: Dr. Mary Flanagan, of the Hunter College Department of Film and Media Studies, and Dr. Andrea Hollingshead, of the University of Southern California Annenberg School for Communication. Dr. Flanagan and Dr. Hollingshead have been collaborators with Dr. Perlin on a project called RAPUNSEL, with which I am fortunate to be involved. Both of these professors have helped me to broaden my horizons and to bridge connections between computer science and its application to other disciplines. A significant amount of my learning from RAPUNSEL was directly applicable to my thesis work. I thank both of these professors for all they have taught me. Additionally, I thank Dr. Flanagan for her participation as a reader on my thesis proposal committee and my doctoral dissertation committee, and I thank Dr. Hollingshead for going out of her way to participate in my doctoral dissertation defense.

From the start of my journey at NYU, I have learned a great deal from many professors and from my fellow students, too. However, this journey could not have begun were it not for the foundation of knowledge afforded me by the University of Delaware's Department of Computer Science and the Rensselaer Polytechnic Institute's Department of Computer Science during my respective pursuits of my Bachelor of Science and Master of Science degrees. Further, I would also like to acknowledge the support I have received from the International Business Machines Corporation. My friends and colleagues at IBM helped me to prepare for the experience that awaited

me at NYU. In particular, I am indebted to the following people for their efforts on my behalf: William Zeitler, Richard Potts, James Noonan, Carla Shultis, Arnie Bendel, Cheng-Fong Shih, Paul Loftus, and Nicholas Carbone.

In addition to those at NYU and IBM, I am very much appreciative of the friends who have come into my life through the years. The support and understanding I received from these friends has been invaluable. I am especially grateful for their friendship through the many times when my PhD program prevented me from seeing them as much as I would have liked. I offer my deepest thanks to Mark Bilson, Daniel Burkhardt, Jim Ground, Yoan Johnson, and Stephen Toth, as well as to the Robinson and McClain families.

I am extremely fortunate to have a family that values education and hard work. I fondly recall the enormous pride my late maternal grandparents, Casper and Theresa Reaves, took in the educational accomplishments of their children and grandchildren. My late paternal grandmother, Theo Robbins, taught me by example that anything is possible when one combines independence, intelligence, and determination. She was an exceptionally strong woman, whose impact on me cannot be measured. Her husband, my grandfather Alexander Robbins, has likewise led by example, not only in his own educational accomplishments, but also through his steadfast efforts to contribute to others' learning. From his volunteer work serving as a tutor in his community, to his support for his children and grandchildren in their formal educations, to his most informal talks around a dining room table, my grandfather has continued to provide me with countless lessons to this day, whether he has known it or not.

My aunts Alice Robbins and Eloise Robbins have both continued to pursue their own educations, earning numerous advanced degrees and designations by going "back to school" after establishing careers. They, like their parents, have taught me by

example to value a life-long commitment to learning. They have also taught me to take risks and not to fear change, and I am extraordinarily grateful to them.

My uncle, Dr. Casper M. Reaves, Jr., has had a most profound influence on me. “Uncle” is only one of the roles Casper has held in my life; he has in turn served unofficially as my big brother, my mentor, my teacher, my advisor, my confidant, my role model, and my friend. In many ways, were it not for Casper, I would not have even entered the field of computer science. It was Casper who sparked my interest in computers, when, in the early 1980s, as a teenager, he created a video game on his Commodore VIC-20 computer. I was still in grade school, but I clearly remember being fascinated by the notion that I could create my own games. As the years went on, much of my early programming experiences were on computers Casper handed down to me. When I was in high school, Casper’s encouragement to broaden my then limited programming knowledge led me to discover that programming was not all about learning codes. I became interested in the science behind the programming. In my endeavor to continue exploring this science at the doctoral level, Casper’s input has proven invaluable. Having earned a PhD himself, he has been able not only to provide exceptional advice and insight about the process, but also to lend an understanding ear and provide much-needed support when I encountered rigorous challenges along the way. His generosity is boundless, and I owe so much of who I am to him. Casper, you truly are my hero.

I am blessed to have an exceptional sister, Kathy Ritchie. She and her husband, Christopher Ritchie, have provided consistent reassurance to me that with or without a PhD, I will still be loved. Their son, my wonderful nephew Stephen Ritchie, has helped to keep me grounded in the knowledge that when all is said and done, I will still be “Uncle Pooh,” and he will still look up to me. In their travels and career

paths, Kathy and Chris have shown me the importance of remaining true to one's heart and of taking risks when necessary.

The two people most directly responsible for my value of education are my parents, Joyce and Wayne Robbins. My parents gave me an early license to pursue my interests and to chart my own path, and they outfitted me with the intellectual and emotional tools to do so. They instilled in me the ability to dream and yet to temper high hopes with humility. They held high expectations, and they entrusted me with the freedom and the responsibility to meet their standards. They allowed me to make mistakes so that I could learn vital lessons, but they never allowed me to create irreparable damage, and they were always there to encourage me to succeed when I did slip. I am entirely grateful for their lessons, their support, and most of all, their love. They have provided me with the proverbial wings with which to fly.

In addition to the extraordinary family I was lucky to be born into, I feel blessed by the family that I have acquired in recent years. My father-in-law, Murray Halbfish, consistently exemplifies the values of life-long learning and risk-taking. Well-established in his career with advanced degrees and professional designations, he nonetheless continually devotes himself to coursework toward additional degrees and designations, all in the interest of learning. I am extremely grateful to Murray for his positive outlook and "never quit" attitude, which I strive to emulate. My mother-in-law, Gayle Halbfish has also demonstrated the value of working hard, and she has shown me the importance of resourcefulness. She has helped me learn to prioritize, and in a demanding graduate program, that is an important skill. My brother-in-law, Michael Halbfish, Esq., has epitomized the virtue of tenacity. His commitment to his profession and his tireless efforts therein have motivated me in my own endeavors. Additionally, Michael's pursuits of outside interests have taught me the importance

of being well-rounded and of pursuing education that addresses the many facets of oneself. Ironically, my “in-laws” do not include the term “in-law” in their vernacular, and as such, they have always treated me as a son and a brother. I am deeply appreciative. For as long as I have known them, they have never ceased to express their affection toward me or pride in me, and the reassurance I have gleaned from them has proven limitless. In addition, their confidence in me has been unwavering and has bolstered me at times when I truly needed it. The love, encouragement, and support they have offered, and the sincerity with which it has been offered, has meant more to me than any mere words on paper could ever express.

No mention of my family would be complete without acknowledgment of my wife, Julie Robbins. Without her, none of this would have been possible. Her support and confidence in me was unwavering. I could not have faced the many challenges of the PhD program without the reassurance that she would be there afterward—to console me if I did not succeed or to celebrate with me if I did. It should be no surprise that academia was also what brought Julie into my life. We met at the University of Delaware eleven years ago and have been inseparable ever since. I am unable to express how much I value and enjoy her companionship. She makes every day more meaningful than I could possibly imagine or explain. Julie, you have my undying love and endless devotion.

Collectively, as noted, my family members have all inspired my value of education, my love of learning, and my abilities to pursue my academic goals. Nonetheless, the chief lesson my family has taught me is that the most important learning one achieves has nothing to do with academics and everything to do with character. Indeed, what we know is of little importance compared to how we use what we know—and degrees of academia cannot substitute for degrees of kindness. My family members have all

instilled in me various lessons and tools to facilitate my academic success, and I will be eternally grateful. Most of all, however, I am grateful to my family members for exemplifying the purest of character. My family has taught me that class is not just about school and grace is not just about movement.

# Abstract

Ongoing improvements to the performance and accessibility of less conventional input modalities such as speech and gesture recognition now provide new dimensions for interface designers to explore. Yet there is a scarcity of commercial applications which utilize these modalities either independently or multimodally. This scarcity partially results from a lack of development tools and design guidelines to facilitate the use of speech and gesture.

An integral aspect of the user interface design process is the ability to easily evaluate various design solutions through an iterative process of prototyping and testing. Through this process guidelines emerge that aid in the design of future interfaces. Today there is no shortage of tools supporting the development of conventional interfaces. However there do not exist resources allowing interface designers to easily prototype and quickly test, via remote distribution, interface designs utilizing speech and gesture.

The thesis work for this dissertation explores the development of an Extensible MultiModal Environment Toolkit (EMMET) for prototyping and remotely testing speech and gesture based multimodal interfaces to three-dimensional environments. The overarching goals for this toolkit are to allow its users to:

- explore speech and gesture based interface design without requiring an understanding of the details involved in the low-level implementation of speech or gesture recognition,
- quickly distribute their multimodal interface prototypes via the Web, and
- receive multimodal usage statistics collected remotely after each use of their

application.

EMMET ultimately contributes to the field of multimodal user interface design by providing an environment to existing user interface developers in which speech and gesture recognition have been seamlessly integrated into their palette of user input options. Such seamless integration serves to increase the utilization within applications of speech and gesture modalities by removing any actual or perceived deterrents to the use of these modalities versus the use of conventional modalities. EMMET additionally strives to improve the quality of speech and gesture based interfaces by supporting the prototype-and-test development cycle through its Web distribution and usage statistics collection capabilities. These capabilities also allow developers to realize new design guidelines specific to the use of speech and gesture.

# Contents

Dedication	iii
Acknowledgments	iv
Abstract	xi
List of Figures	xvi
List of Tables	xx
List of Appendices	xxi
<b>1 Introduction</b>	<b>1</b>
1 Multimodal Interface Design . . . . .	4
2 Human Factors in Speech-Based Interfaces . . . . .	6
<b>2 Multimodal Interface Design History and Related Work</b>	<b>9</b>
1 Early Multimodal Interfaces . . . . .	9
1.1 Put-That-There . . . . .	10
1.2 CUBRICON . . . . .	12
1.3 XTRA: An Intelligent Multimodal Interface to Expert Systems	15

2	Recent Speech Based Multimodal Interfaces . . . . .	19
2.1	QuickSet . . . . .	20
2.2	IBM's Human-Centric Word Processor (HCWP) . . . . .	23
2.3	Portable Voice Assistant . . . . .	25
2.4	Field Medic Information System . . . . .	27
3	Multimodal Interface Usability Studies . . . . .	28
3.1	Human Factors in Integration and Synchronization of Input Modes . . . . .	28
3.2	Mutual Disambiguation of Recognition Errors . . . . .	33
3.3	Multimodal Interfaces for Children . . . . .	35
4	Multimodal Interface Design and Related Work Conclusion . . . . .	37
5	Multimodal Frameworks Currently in Development . . . . .	39
<b>3</b>	<b>EMMET Design and Implementation</b>	<b>42</b>
1	Overview . . . . .	42
2	Foundational Concepts and Technologies . . . . .	43
2.1	Java Speech API (JSAPI) . . . . .	44
2.2	HHReco Graphic Symbol Recognition Toolkit . . . . .	47
2.3	The jMonkeyEngine (jME) Java 3D Rendering API . . . . .	47
2.4	Unification-based Multimodal Integration . . . . .	48
2.5	Statistics Gathering . . . . .	50
3	EMMET's Software Architecture . . . . .	51
3.1	The <i>content</i> Package . . . . .	52
3.2	The <i>controller</i> Package . . . . .	60
3.3	The <i>command</i> Package . . . . .	103

3.4	The <i>resolver</i> Package . . . . .	114
3.5	The <i>model</i> Package . . . . .	118
3.6	The <i>statistics</i> Package . . . . .	121
3.7	The <i>apps</i> Package . . . . .	124
<b>4</b>	<b>EMMET Proof of Concept Tests, Function Verification Tests, and Demonstration Applications</b>	<b>126</b>
1	Pre-EMMET Proof of Concept Tests . . . . .	126
1.1	Early Speech Recognition Tests . . . . .	127
1.2	Speech and Gesture Based Geometry Placer . . . . .	130
2	EMMET Development Function Verification Tests . . . . .	132
2.1	EMMET Unimodal Input Mode Test Applications . . . . .	134
2.2	EMMET Multimodal Input Mode Test Application . . . . .	142
3	EMMET Geometry Placer Demonstration Application . . . . .	147
<b>5</b>	<b>Conclusion</b>	<b>157</b>
	<b>Appendices</b>	<b>160</b>
	<b>Bibliography</b>	<b>164</b>

# List of Figures

2.1	Bolt’s Media Room . . . . .	10
2.2	CUBRICON System Architecture . . . . .	13
2.3	XTRA Tax Form . . . . .	16
2.4	Spatial Relation . . . . .	18
2.5	Quickset Handheld PC Interface . . . . .	21
2.6	Quickset Gestures and Symbols . . . . .	22
2.7	IBM’s HCWP Input Evaluation System . . . . .	24
2.8	VoiceLog Interface . . . . .	26
2.9	Task Action Command Analysis . . . . .	30
2.10	Mutual Disambiguation Research Design . . . . .	33
3.1	JSGF Sample . . . . .	45
3.2	JSGF “Put-That-There” Sample . . . . .	46
3.3	Gesture Feature Structure . . . . .	49
3.4	Speech Feature Structure . . . . .	49
3.5	Generated Multimodal Feature Structure . . . . .	50
3.6	<i>content</i> Package Overview . . . . .	53
3.7	ModalContent Class . . . . .	54
3.8	ModalContent Subclasses in <i>controller</i> Package . . . . .	55

3.9	ModalityType Class . . . . .	56
3.10	ModalitySubType Subclasses in <i>controller</i> Package . . . . .	57
3.11	TimeInterval Class . . . . .	58
3.12	ConfidenceLevel Class . . . . .	60
3.13	ModalContentQueue Class . . . . .	61
3.14	Gesture Input Handler and Adapter . . . . .	67
3.15	Gesture Content subclass . . . . .	69
3.16	KeyboardInputManager Class . . . . .	71
3.17	KeyContent Class . . . . .	72
3.18	KeyHandler Interface and KeyAdapter Class . . . . .	73
3.19	MouseInputManager Class . . . . .	75
3.20	MouseContent Class . . . . .	76
3.21	Mouse and Mouse Motion Input Handlers and Adapters . . . . .	77
3.22	<i>controller.gesture</i> Package . . . . .	79
3.23	Gesture Class . . . . .	81
3.24	GestureInputManager Gesture Recognition State Cycle . . . . .	82
3.25	The <i>controller.speech</i> Package . . . . .	86
3.26	SpeechInputManager Class . . . . .	88
3.27	ParsedRuleToken Class . . . . .	89
3.28	ParsedRuleNode Class . . . . .	89
3.29	Simple JSGF Grammar for Adding and Coloring Objects . . . . .	90
3.30	ParsedRuleNode Tree . . . . .	90
3.31	ParseRuleUtil Utility Class . . . . .	92
3.32	RuleContent Class . . . . .	93
3.33	RuleHandler Interface . . . . .	94

3.34	MultimodalInputManager Class . . . . .	100
3.35	Multimodal Input Listener and Adapter . . . . .	102
3.36	ModeTrigger Interface and Implementors . . . . .	105
3.37	CommandDefinition Class . . . . .	106
3.38	The CommandListener Interface and CommandAdapter Class . . . . .	109
3.39	The CommandEvent Class . . . . .	111
3.40	The CommandDefinitionRegistry Class . . . . .	113
3.41	The Resolver Class . . . . .	117
3.42	WorldObject Class with Mobile and Static Subclasses . . . . .	119
3.43	WorldObject and WorldObjectType Registries . . . . .	120
3.44	MultimodalStatisticsCollector . . . . .	123
3.45	CommandCallRecord . . . . .	124
3.46	CommandStatistics Class . . . . .	125
4.1	PollyWorld Layout Tester Java Applet with Speech Recognition . . . . .	129
4.2	Speech and Gesture Based Geometry Placer Proof-Of-Concept . . . . .	133
4.3	TestKeyboardInput Excerpt . . . . .	135
4.4	TestMouseInput Excerpt . . . . .	136
4.5	TestGestureInput Excerpt . . . . .	137
4.6	TestGestureInput Application with User Drawing a Tree Gesture . . . . .	138
4.7	TestSpeechInput Excerpt . . . . .	140
4.8	Basic Speech Definition Use of Command and Trigger . . . . .	143
4.9	Look Left Definition Responds to Speech or Keyboard Triggers . . . . .	144
4.10	ColorSomething Speech Definition Responds to Multimodal Input . . . . .	145
4.11	EMMET Geometry Placer . . . . .	148

4.12	Registering World Object Types and the WorldObjectRegistry . . . .	149
4.13	Geometry Placer “addNewObject” CommandDefinition Triggers . . . .	150
4.14	Geometry Placer “addNewObject” CommandDefinition Listener . . . .	152
4.15	EMMET Geometry Placer Java™ WebStart JNLP File . . . . .	154
4.16	EMMET Geometry Placer Remote Statistics Reporting . . . . .	155
4.17	Geometry Placer “addNewObject” Command Usage Statistics . . . .	156

# List of Tables

2.1	Types of Individual Input Construction . . . . .	29
2.2	Pen Input Command Type Frequencies . . . . .	32
2.3	MD Rates and Command Length . . . . .	34

# List of Appendices

Appendix A: Glossary of Terms . . . . . 160  
Appendix B: UML Class Diagram Reference . . . . . 163

# Chapter 1

## Introduction

Multimodal interfaces process two or more combined user input modes in a coordinated manner with multimedia system output. Such combinations work to facilitate the overall human computer interaction experience. There is a growing interest in the design and implementation of multimodal interfaces fueled by the many inherent advantages they provide. Multimodal systems are flexible in their ability to provide users with a choice of input. They offer greater accessibility to a broad range of users. Their adaptability is apparent in their ability to switch input modes as necessary. The simultaneous input possibilities afforded by multimodal interfaces allow for more efficient input. Multimodal systems can also take advantage of mutual disambiguation to facilitate error avoidance and recovery.

In a summary of future directions in multimodal interfaces Oviatt et al. specify a number of research challenges involved in advancing the field [46]. One of these challenges is the scarcity of tools which facilitate the development of multimodal software and the inherent complexity in the development of such tools. Tools such as these should allow for the rapid implementation of multimodal interfaces

for iterative software development. A second challenge includes the development of metrics and techniques for evaluating alternative multimodal systems. These metrics and techniques are needed to establish a common basis for analyzing the quality of new multimodal interface designs. A third challenge involves studies of how people integrate and synchronize multimodal input during human-computer interactions. Such empirical usability studies explore and evaluate the human factors involved in multimodal input, thus providing useful insight and guidance toward the design and implementation of future multimodal interfaces.

To address the challenges posed by Oviatt et al., this dissertation explores the design and implementation of EMMET, an Extensible MultiModal Environment Toolkit. EMMET was created with three overarching goals:

- to allow programmers to explore speech and gesture based interface design without requiring an understanding of the details involved in the low-level implementation of speech or gesture recognition;
- to facilitate rapid distribution of programmers' multimodal interface prototypes via the World Wide Web; and
- to provide a built in data collector that enables programmers to receive multimodal usage statistics after each use of their applications.

Note that for each goal there exists a corresponding implicit research question of how one can develop a programmer interface to accomplish the goal. Furthermore, reaching these goals supports the thesis that it is possible to implement a clean and well architected set of algorithms and techniques to support the development and evaluation of multimodal interfaces by handling the difficult semantic problems that

arise when trying to reconcile very different input modalities such as speech and gesture.

Thus, EMMET first contributes to the field of multimodal interface design by addressing Oviatt et al.'s call for tools that facilitate the development of multimodal software. EMMET enables programmers who lack expertise in speech and gesture recognition to create applications with speech and gesture based multimodal interfaces. As a result, using EMMET to implement speech and gesture recognition, programmers need only to define the grammar string and gestures they want their applications to recognize and specify what response is called for when a qualifying utterance or gesture occurs.

Additionally, EMMET responds to Oviatt et al.'s discussion of the need to develop metrics and techniques for evaluating alternative multimodal systems. Each time a program that was created with EMMET is used, EMMET collects multimodal usage statistics and sends them to the programmer. Realizing that the quality of multimodal interface designs will inevitably vary based on the differing purposes of each application, the creation of one rubric by which all multimodal interfaces can be evaluated would be daunting. However, EMMET allows programmers to independently evaluate their multimodal interfaces and customize them to their needs and the needs of their users. Therefore, EMMET opens a door toward answering Oviatt et al.'s third challenge, the call for studies of how people integrate and synchronize multimodal input during human computer interactions. From the information EMMET provides, developers will be able to gain insight and guidance for the design and improvement of future multimodal interfaces.

# 1 Multimodal Interface Design

The primary goal in the design of any user interface is to facilitate the interaction between user and machine. This user-centered goal is the guiding force behind choices made in the design process. There are, of course, many system engineering issues that influence interface design decisions such as the limits of technology, schedules, proper functionality, reliability, etc. However, addressing these issues ideally serves the purpose of creating a better user experience with the system.

One purpose of researching multimodal interfaces from an HCI perspective is to evaluate how to take advantage of the benefits they provide over unimodal recognition-based interfaces and conventional keyboard and mouse interfaces. Such advantages include flexibility, availability, adaptability, efficiency, lower error rate, and a more intuitive and natural interaction [45].

The flexibility of multimodal interfaces lies in the choices they give to users in selecting how they will interact with the system. These choices allow the user to select an input mode to suit the type of input, to use multiple input modes simultaneously, or to alternate between modes. An example of such flexibility is evident in the Field Medic system (section 2.4), as it allows a medic to alternate between using voice, pen, or both as necessary. This provides the medic with a hands free interface whilst he or she cares for the patient and the ability to later switch to a pen and tablet based interface for recording more detailed information at a later time.

The ability of multimodal interfaces to accommodate a broad range of users under a variety of circumstances increases the availability of such interfaces versus the availability of conventional unimodal interfaces. This increased availability can extend access to a particular system to users that may otherwise not be able to interact

at all. This is self evident for users such as the visually impaired. However to argue that a unimodal speech-based interface serves the same purpose overlooks the fact that while speech alone suits the needs of a blind user, it prevents use by one that is hearing impaired. The advantage of multimodality is that one interface can allow a range of users to interact with the same system.

Multimodal interface adaptability is an application of flexibility in which a multimodal system can be taken into an environment where one form of input becomes unusable and yet the system can still be used by switching to an alternative input method. An example situation for adaptability would be a vehicle navigation system which supports visual directions via a map display and directions delivered by speech. In the common situation in which the driver cannot remove his gaze from the road, a purely map-based interface would be unable to direct him; however listening to verbal driving directions is still an option. A similar example is a cell phone that allows the owner to use spoken name recognition for dialing common numbers.

A greater efficiency for certain tasks is gained with multimodal interfaces that process parallel input [34]. These tasks include those that would require a series of sequential input events in a unimodal interface. This ability to process parallel input was originally thought to be the primary advantage of using multimodal interfaces [45]. One example of improved efficiency is experienced by users of the map-based LeatherNet military simulation system (section 2.1), as it allows them to quickly specify objects by name while simultaneously indicating an action using stylus input.

Error handling is also improved in multimodal systems due to their inherent error avoidance and recovery. This avoidance and recovery is often initiated by the user as they tend to select input modes that they believe best suit content they are about to enter. Also, in pen and gesture based multimodal input, the ability to select a visible

referent rather than having to use speech avoids possible errors in speech recognition. Finally, users will often base mode type selections on past experience, such that past errors that occurred when using a particular input mode are avoided when the user needs to enter a similar input [45]. In addition to user originated error avoidance and recovery, system based error detection and recovery is also improved in multimodal interfaces. This improvement is rooted in the ability to use mutual disambiguation when processing parallel complimentary recognition inputs. An example of such disambiguation occurs when a spoken referent phrase is misrecognized as being singular, but the user has used pen input to select multiple items on a display.

## **2 Human Factors in Speech-Based Interfaces**

In discussing the human factors of speech as it relates to speech based interface design it is important to differentiate between interfaces that recognize command speech versus those that recognize natural language. Command speech involves the recognition of token words or phrases in isolation, while natural language involves the recognition of the full sentence structures and richness of a natural spoken language such as English, although perhaps with a limited vocabulary. Individual voice quality contributes to the difficulty of recognizing speech tokens. Interpersonal variations in voice and speech are caused by such differences as physical features, cultural speech patterns, gender, native language, and accents. Intrapersonal variability is affected by differences like time of day, health, and degree of background noise. To facilitate recognition when faced with this variability, systems must take advantage of syntax, semantics, and context. The accuracy of command speech recognition can be increased by maintaining a current context that limits the possible valid commands.

An advantage in natural language recognition is that much of the speech input beyond simple command tokens provides helpful contextual information and redundancy.

Human short term memory is another concern in designing a speech based interface. Most humans can keep about five to nine chunks of information in working memory at a time [51]. This factor is not as important for visual GUIs, because they limit the user to selecting or acting upon a shown set of choices. However, similar to command line interfaces, the efficient use of command speech interfaces involves remembering the set of valid inputs tokens or else having to repeatedly refer to help systems or documentation. In purely speech based interfaces a guidance system, similar to the one inherent in GUI systems, needs to be provided.

In discussing speech based interface design dialog structure must also be considered. Conventional user interface commands are usually short, as they are often based upon a single word or mouse input event. With spoken dialog there is the possibility for longer input streams, especially when dealing with natural language. The question of when to respond to user input is difficult because the definite completion of input declared by pressing enter on the keyboard or clicking a mouse button is lost when dealing with speech.

Conversational technique is another human factor to address in the development of speech-based interfaces. Techniques such as clarification, back-channel utterances, dialog repair, turn taking, and topic introduction commonly occur in human-human conversation [2]. These techniques often produce problems that do not have conventional user interfaces counterparts. Back-channel utterances include non-speech sounds such as “uh-huh”, “um”, and “ah”. Dialog repair involves a pause in the current flow of speech to immediately step back and correct a pronunciation or to change a word or phrase. Regarding turn taking, human-computer interaction is based on

the user producing a complete utterance or phrase followed by a suitable response from the computer. Human-human turn taking is variable, allowing one party to produce sequential complete utterances with no need for intermittent responses from the other. Sometimes the meaning of one phrase is clarified by a later one. Similarly, a topic introduction can change the context of subsequent dialog such that the interpretation of a phrase becomes completely different from its interpretation prior to the introduction.

User vocabularies vary in size and breadth but the only vocabulary of value to them is the one recognized by the interface. How to establish an interface's vocabulary with the user is an important question in the development of both command speech and natural language interfaces. One solution is to rely upon a phenomenon called "parroting", the human tendency to respond when spoken to using the same vocabulary and style as the speaker. Thus, restricting system prompts and responses to the supported vocabulary may influence the users input vocabulary.

Speech prosody is another human factors consideration. Prosody refers to variations in the duration of phonemes and the silences between them as well as frequency and amplitude changes in human speech patterns. Prosody is used by natural language systems to improve recognition and parse information. Prosody is also used by interfaces that generate speech to produce more natural sounding output.

# Chapter 2

## Multimodal Interface Design

### History and Related Work

The following chapter surveys a number of early and recent multimodal interfaces as well as three multimodal interface usability studies. Researching these interfaces and usability studies later served to guide related aspects of the design and implementation of EMMET.

#### 1 Early Multimodal Interfaces

One of the earliest multimodal interfaces illustrating the use of voice and gesture based input was Richard Bolt's "Put That There" system [1]. Subsequent multimodal interfaces of the late 1980's and early 1990's explored the use of speech input combined with conventional keyboard and mouse input. The design of these interfaces was based upon a strategy of simply adding speech to traditional graphical user interfaces (GUIs). The primary motivation for this addition of speech was a belief that the use

of speech gives the user greater expressive capability, especially when interacting with visual objects and extracting information [45]. Examples of such types of interfaces include CUBRICON [38], XTRA [57], and Shoptalk [10].

## 1.1 Put-That-There

In Bolt's "Put-That-There" system, speech recognition is used in parallel with gesture recognition. User interaction takes place in a media room about the size of a personal office as seen in Figure 2.1.

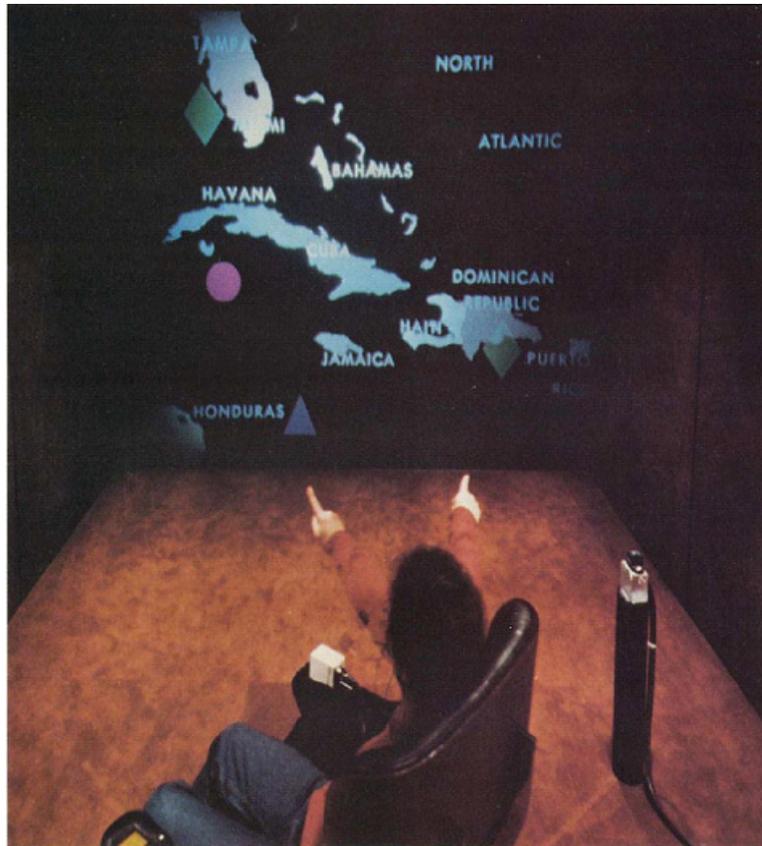


Figure 2.1: Bolt's Media Room

Visual focus is directed at a large screen display on one wall of the room. Gesture-

based input is primarily the recognition of deictic arm movements in the forms of pointing at objects displayed on the screen and sweeping motions of the arm whilst pointing. In general, deictic gestures are gestures that contribute to the identification of an object (or a group of objects) by specifying their location. The gesture recognition technology Bolt used involves a space position and orientation sensing technology based on a magnetic field [1]. Speech recognition in the “Put That There” system allows for simple English sentence structures that use a limited vocabulary. The driving example scenario Bolt uses to illustrate his “Put That There” system consists of input requests for creating, customizing, copying, moving, and deleting basic geometric objects on a large screen display. Sample input speech includes the command, “Create a blue square there.” The difficulty of interpreting this request is the presence of the pronoun “there.” In a purely speech based interface the specification of a location must be included as part of the speech command. For example, after uttering, “Create a blue square,” the user must provide location information in the form of a phrase such as, “in the center of the display,” or perhaps, “next to the green circle,” (assuming referable objects, such as a green circle, exist). Bolt’s system addresses the ambiguity of deictic references by assigning pronouns to temporal arm pointing and motion gestures. Such gesture-based recognition builds upon the speech input to disambiguate pronoun usage. Thus, when the user states, “Make that smaller,” while pointing to a blue square, the pronoun, “that” refers to the blue square. A more ambiguous command is the paper’s title providing phrase, “Put that there.” For this phrase, the user points at an object while saying “that,” then points at a desired location after saying “there.” This disambiguation represents one way multimodal interfaces can cooperatively use one modality in parallel with another. Accordingly, Bolt introduced an important benefit of multimodal interfaces by demonstrating their

ability to resolve deictic references that unimodal interfaces cannot consider.

The speech utterances recognized by Bolt's "Put-That-There" system are limited to its set of command words. A limited speech recognition vocabulary can be useful because it improves recognition efficiency and accuracy [17, 33]. Such limitation of input also occurs in Windows-Icon-Menu-Pointer (WIMP) based interfaces, in which the user is limited to referring to a finite set of command choices, windows, and icons.

Bolt's system provides an initial step in establishing multimodal interfaces as a more natural form of human-computer interaction. This is especially evident in the user's ability to use pronouns as would occur in daily conversation, and the natural manner of pointing to an object to establish it as the subject of current discourse or as the antecedent of a pronoun. A direct implication of Bolt's work on the design of EMMET was the guiding principle of deictic reference resolution. Based on Bolt's research, one of the goals in developing EMMET was to provide a system that would allow programmers to create interfaces that could resolve deictic references. Another way Bolt's system directly impacted the development of EMMET was in its use of a limited, pre-defined vocabulary. Programmers who utilize EMMET provide grammar strings to specify speech utterances to which they want their interfaces to respond.

## **1.2 CUBRICON**

An early interface combining spoken and typed natural language with deictic gesture for the purposes of both input and output was designed for CUBRICON [39], a military situation assessment tool. Similar to the "Put-That-There" system, the CUBRICON interface utilizes pointing gestures to clarify references to entities based upon simultaneous natural language input. It also introduces the concept of composing and generating a multimodal language based on a dynamic knowledge base. This

knowledge base is initialized and built upon via models of the user and the ongoing interaction. These dynamic models influence the generated responses and affect the display results which consist of combinations of language, maps, and graphics.

In the CUBRICON architecture (Figure 2.2), natural language input is acquired via speech recognition and keyboard input. Location coordinates are specified via a conventional mouse pointing device. An input coordinator processes these multiple input streams and combines them into a single stream which is passed on to the multimedia parser and interpreter. Building upon information from the system's knowledge sources, the parser interprets the compound stream and passes the results on to the executor/communicator.

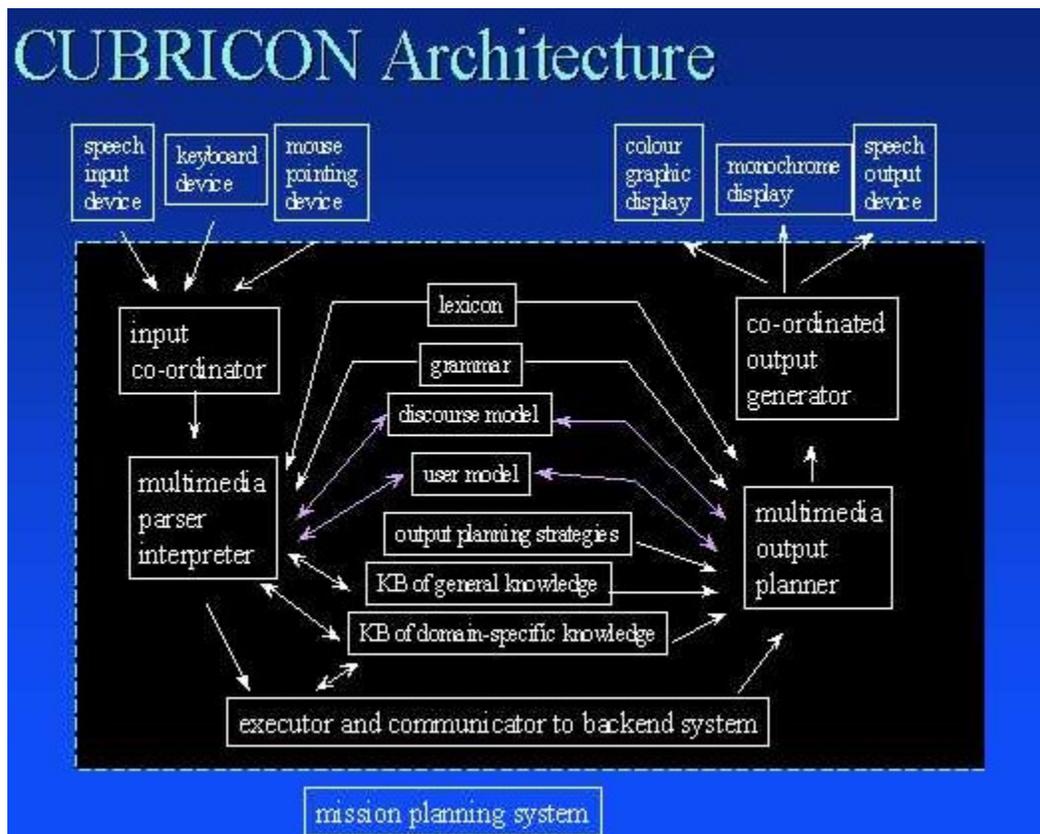


Figure 2.2: CUBRICON System Architecture

The CUBRICON system's knowledge sources consist of the following elements:

- Lexicon - denotes the vocabulary
- Grammar - defines the multimodal language
- Discourse Model - dynamically maintains knowledge pertinent to the current dialog.
- User Model - aids in interpretation based on user goals and plans
- Knowledge Base - contains information related to the task space

CUBRICON's multimodal language incorporates mouse input to select on-screen content such as windows, table components, icons, and points, and spoken or written natural language, to specify one or more actions that refer back to selected objects. CUBRICON builds upon "Put-That-There" by allowing a number of point gestures in a single phrase and the combination of multiple multimodal phrases into one sentence. For instance, CUBRICON allows one to use a phrase like, "Where are these items?" while sequentially pointing to multiple elements.

The combination of speech and gesture in this manner improves the usability of either input method alone, as the two can work cooperatively to achieve greater accuracy in determining the user's intent. Thus the interpretation of an ambiguous utterance can take advantage of the fact that only a limited set of applicable actions exist for the referenced object. Conversely, an ambiguous pointing gesture can be resolved if simultaneous natural language input reduces the number of applicable on-screen objects.

CUBRICON's output is also multimodal, as it integrates gesture with speech. For instance, if an output refers to an icon object, the icon referenced is pointed to,

and corresponding natural language is generated. If the output object is part of an icon, the containing icon is pointed to instead. If the output refers to an object that appears in multiple windows, the object is weakly highlighted in each window, except for the top or selected window, in which the icon blinks.

By definition, multimodal output methods, like the one exhibited by the CUBRICON interface, exceed conventional output methods' abilities to clearly convey information to the user in a natural manner ("natural" denoting similarity to human-human interaction). An illustrative example is the explanation of driving directions, in which a graphic map traversal combined with verbal directions is clearer than that of either output alone.

The explanation of CUBRICON's architecture provided guidance in architecting EMMET. Similar to the CUBRICON architecture, EMMET maintains a grammar for deriving recognizable utterances. Further, EMMET maintains both a discourse model, in the form of content registries, and a knowledge base, in the form of user-defined command tasks.

### **1.3 XTRA: An Intelligent Multimodal Interface to Expert Systems**

XTRA (eXpert TRAnslator) is an intelligent multimodal interface that combines natural language, graphics, and pointing for input and output [57]. Based upon a focusing gesture-analysis methodology, the XTRA project constrains referents in speech to possibilities from a gesture based region. Doing so aids the system in interpretation of subsequent definite noun phrases that refer to objects located in the focused area.

An illustrative application discussed by Wahlster involves the use of XTRA to facilitate filling in a tax form. As shown in Figure 2.3, gesture based input and output for this application occurs in the left panel which displays pages of a tax form. Natural language input and system response text are displayed in a panel to the right. Note that the tax form display panel is shared for both gesture based input and output.

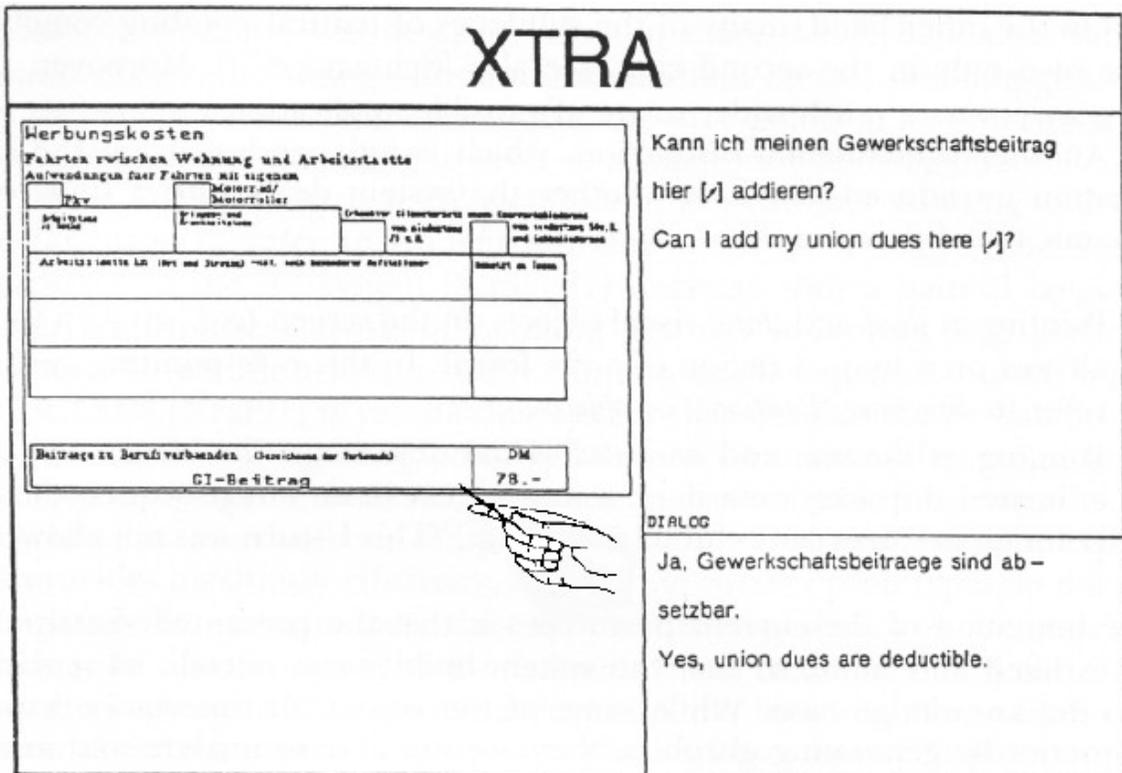


Figure 2.3: XTRA Tax Form

Using a mouse or similar pointing device, the user can specify locations on, and areas of, the tax form. Fields that exist on a tax form page may overlap or be contained within one another. Also, analogous to human-human interaction, but unlike conventional human-computer interaction, gestured-to locations are not confirmed

graphically. The granularity and interpretation of mouse-specified locations and areas depends upon the current pointing mode selected by the user. These modes are designed to simulate various types of deictic gestures commonly used in human-human conversation, as follows:

- Exact pointing with a pencil
- Standard pointing with the index finger
- Vague pointing with the entire hand
- Encircling regions with an '@'-sign

In addition, three types of movement gestures are considered: point, underline, and encircle. Selecting in pencil mode is similar to mouse selection in conventional WIMP-based interfaces; however, as the pointing area mode becomes less granular, mouse selections are no longer considered to occur in discrete fields. Instead, a plausibility value is computed for each subset of the superset generated with all of the fields contained in the pointing-mode based mouse selection region. Thus a selection of multiple tax form fields as a referent could be accomplished by using the entire hand mode and using plurality in the natural language discourse.

Also, XTRA considers the effects of dialog focus, thereby allowing the user to sequentially or simultaneously specify a region to be the one containing another location or area. Allowing the user to do so removes the ambiguity of language based references to a field that may occur in multiple locations. For instance, if field 'A' is verbally specified but it appears in three rows, a focusing gesture can be used to specify the row of attention. The discussion of dialog focus progresses toward a

consideration of combined pointer actions in which one pointer action specifies a region and another indicates a subfield of that region. In the case of pointer actions, two possibilities are considered: one-handed input (sequential), and two-handed input (parallel/simultaneous). In this manner, focusing gestures modify the discourse model.

The discourse model can also be modified based upon intrinsic and extrinsic relations. Ignoring the pointing icon in Figure 2.4, the definite noun description, “the right circle” is intrinsically determined to be circle 5, based upon the standard left to right reading direction paradigm. However, if one were to consider the presence of the pointing icon, the same description would extrinsically be determined to refer to circle 3.

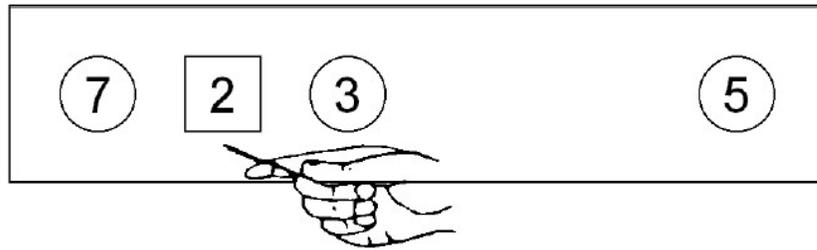


Figure 2.4: Spatial Relation

Because multimodal interfaces involve the process of interpreting multimodal input as well as generating multimodal output, XTRA’s output consists of simultaneous deictic descriptions and pointing gestures. In planning the presentation of such output, XTRA consults its dynamic user model to customize responses based upon the user’s experience level. For example, a pointing gesture is used in place of a verbal description to refer to a visible object if a technical term would be required that is not understood by the user, based upon the current user model. XTRA also utilizes

the different types of pointing modes made available to the user for output. This feature helps to clarify the scope of an output utterance. For instance, when an output utterance is, “Delete this,” and pencil mode is used to point to a word phrase, it is hard to determine if the deletion affects the exact letter pointed to by the pencil, or the whole phrase. However, if index pointing mode is used, the user’s index pointing gesture indicates whole words or fields, clarifying what the deletion will affect.

XTRA is an early illustration of how multimodal input may affect dynamic user models and dialog discourse models, and how such models may affect the multimodal output of a cooperative natural language and gesture based interface. In addition, XTRA introduces the use of deictic gesture granularity to parallel natural gesture usage in human-human interaction. XTRA also shows a use of sequential or simultaneous pointing gestures, in which one gesture establishes an area of attention to reduce or remove ambiguity in another gesture.

XTRA’s implication on the development of EMMET is evident in EMMET’s ability to recognize non-symbolic gestures and, upon recognition of such gestures, to provide programmers using the EMMET API with pertinent attributes of the gesture. Subsequently, these programmers may analyze the gesture attributes to mimic deictic granularity demonstrated by Wahlster’s system.

## **2 Recent Speech Based Multimodal Interfaces**

Recent multimodal interface trends have moved away from combining speech with simple mouse and touchpad pointing, and toward the use of speech in parallel with more expressive input methods and technologies [45]. Such recent interfaces benefit from the additional user expressibility allowed by two recognition based inputs. Cur-

rently the most mature research in multimodal interfaces, combining two recognition based inputs, has focused on speech and pen or speech and lip recognition. For both cases keyboard and mouse input tends to not be used.

## 2.1 QuickSet

Research into speech and pen based multimodal input began in the early 1990's. The QuickSet system, prototyped in 1994, is one of the earliest speech and pen multimodal interfaces [45]. Quickset is a collaborative multimodal system designed to run on multiple platforms from handheld PCs (see Figure 2.5) to wall-sized display interfaces. In addition to integrating multiple interface components, the Quickset system is designed to work with a collection of distributed applications [13]. A Java-based implementation of Quickset was developed for the World Wide Web. The system also introduces a unification-based mechanism to analyze the meaning of multiple input mode fragments. This mechanism selects the optimal joint interpretation of sequential or simultaneous input fragments. Like the CUBRICON and XTRA systems, Quickset utilizes multimodal discourse to aid in accurate interpretation of speech and gesture input.

Quickset [13] is designed as a general architecture for providing speech and pen multimodal interfaces for map-based, otherwise self contained, back-end applications. The map interface provided by QuickSet displays the terrain for a specified region along with entities whose physical position lies within the region. Normal map interface capabilities such as zoom and pan are also provided. Multimodal pen and speech input allows the user to annotate the map using points, lines, and areas. The user can also use symbolic gestures to create new entities on the map while simultaneously using speech input to describe and name them. To handle the situation where



Figure 2.5: Quickset Handheld PC Interface

background conversation or speech is not intended for the interface, QuickSet only activates its speech recognition engine when the pen touches the display. The commercial speech engines used by QuickSet to implement speech recognition, are IBM's VoiceType, a predecessor to the current IBM ViaVoice series, and Microsoft's Whisper engine. The pen-based gesture recognizer was written as part of the QuickSet implementation and consists of a neural network and a set of hidden Markov models. The gesture recognizer recognizes a number of pen gestures including military map symbols, editing gestures, paths, areas, and taps. QuickSet also provides distributed system support, speech recognition customization parameters, and multi-user collaboration.

An early illustrative system using QuickSet as its multimodal interface is LeatherNet, a map based military simulation set-up and control application [7]. With LeatherNet, military units and objectives are placed on the map using speech to specify the object and gesture to specify the locations. New entities can be created

on the map by using line gesture based symbols, as shown in Figure 2.6. These symbolic gestures can be used to create objects like barbed wire, minefields, and platoons, which can later be identified by a name given in speech input as the item is created.

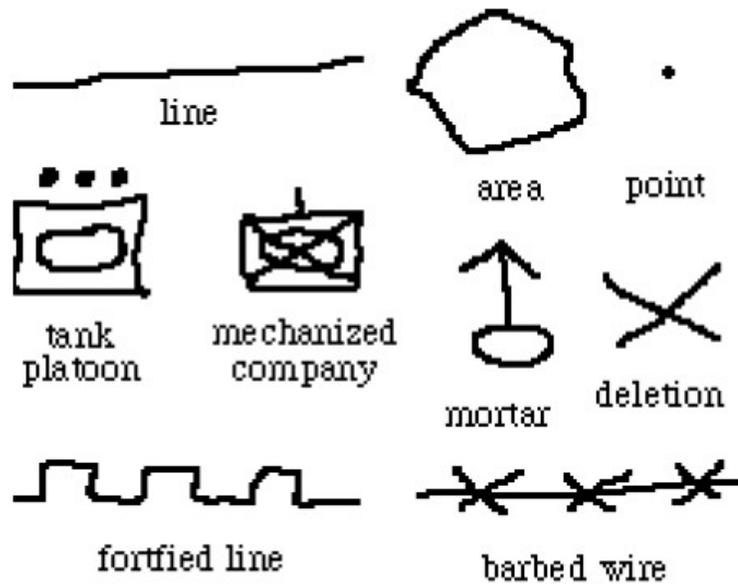


Figure 2.6: Quickset Gestures and Symbols

LeatherNet also utilizes the Quickset framework’s support of heterogeneous collaboration. Thus users can be interacting in the same simulation using variegated platforms and software environments from handheld PCs to PC Web-based applets to wall sized virtual reality 3D caves. In each case the interface displays the same information in whatever form is available for the platform. The collaboration supported includes the ability for each user’s device to display changes in unit and object location by subscribing to an entity-location and attribute database. With the framework users can also choose to link their interfaces with other ones such that they can actually see the pen gestures and map view changes produced by another user.

To conclude, a general framework such as QuickSet facilitates multimodal inter-

face design research by providing a flexible testing environment, in which multimodal interfaces can be developed and refined using a rapid implementation and test cycle. This environment allows researchers to acquire a better understanding of which interface modes and combinations work best with particular application paradigms. The environment also helps researchers determine ways in which various input methods can reduce their weaknesses by taking advantage of other input methods' strengths when included in a multimodal interface.

The EMMET API builds upon the functionality provided by Quickset. Multimodal interface capabilities supported by Quickset, such as multi-stroke symbolic gesture recognition and non-symbolic gesture recognition, deictic reference resolution, and modal speech recognition, are all provided by EMMET. In addition, applications developed using the EMMET API are not restricted to map-based environments. Furthermore, EMMET is not restricted to two-dimensional application environments, as it supports and provides additional functionality for multimodal interfaces to three-dimensional environments.

## **2.2 IBM's Human-Centric Word Processor (HCWP)**

IBM's Human-Centric Word Processor (HCWP) is a word processing system that allows content to be input via continuous real-time dictation, then edited using a pen and speech based interface (Figure 2.7). HCWP was built to address the common desire to correct and edit text obtained via speech dictation [56]. This desire to edit is often due to a lack of organization common in real-time dictation and inaccurate speech-recognition [46]. To allow such multimodal interactivity HCWP builds upon technologies including speech recognition with natural language processing(NLP) and stylus pen gesture recognition.

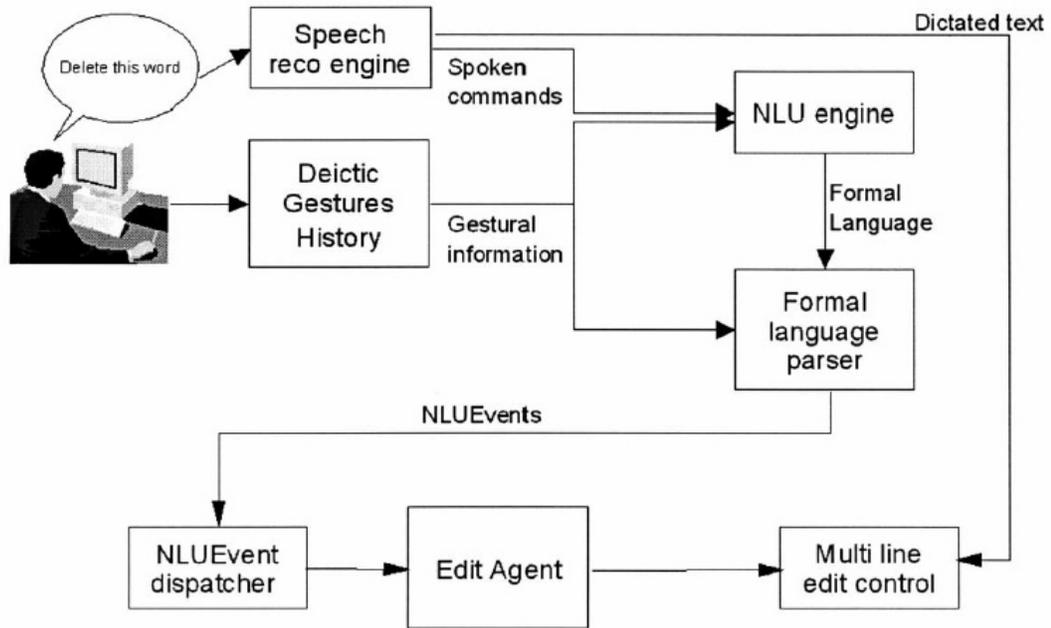


Figure 2.7: IBM's HCWP Input Evaluation System

The post-dictation text editing phase allows the usual keyboard based editing input such as spelling corrections and word changes. Spatial text editing allows for multimodal pen input to format, correct, and manipulate the dictated text. Examples of speech and pen based formatting and correction of text include spoken phrases such, as "Delete this word," whilst pointing to a word in the text, and "Underline from here to there," while indicating the beginning and end of the desired text with sequential deictic pen gestures. Usability test results showed that such multimodal capability reduced task completion time relative to speech only text editing interfaces [56].

To address the common problem of determining whether the user is dictating text or issuing a text editing command, HCWP used a modal rocker-switch on the microphone. Tests showed that users quickly adopted the use of this switch with minimal mode errors. Alternative solutions to this problem, used in similar speech-

based interface situations, include GUI buttons or speech commands to toggle the mode, or the development of a more sophisticated modeless system which determines whether user speech is a command or dictated text based upon its content. Although the modeless approach is considered optimal, studies have shown that errors occurring in such systems tend to iteratively lead to interpretation problems and lack of usability [28].

HCWP accomplishes integration of deictic pen gestures and command speech by maintaining a temporal buffer which indicates the time and location of each gesture. The natural language understanding (NLU) engine uses similarly time-stamped sub-elements of input speech to determine an appropriate formal language statement to pass onto the Formal Language Parser. The parser then builds upon current focus and discourse information to generate events which are dispatched to an editor agent to carry out.

A couple of HWCP aspects influenced design decisions made in implementing the EMMET API. Firstly, the benefits of allowing users to toggle speech recognition on and off, via various methods, resulted in EMMET's push-to-talk mode option and the ability to specify the toggle indicator. EMMET also uses aspects of the HWCP temporal buffer solution for recording unimodal input events.

### **2.3 Portable Voice Assistant**

The Portable Voice Assistant (PVA) by BBN is a pen and speech recognition based architecture for developing multimodal interfaces for on-line Web applications. PVA runs on a mobile pen-based computer with microphone and wireless connection to the Internet [32]. VoiceLog is a prototype system implemented as a Java Web-based applet to demonstrate the PVA interface. The VoiceLog applet allows the user to

order parts from a catalog. As illustrated in Figure 2.8, the VoiceLog applet consists of a small status field above a large display panel which shows either parts from the catalog or an order form. Images from the catalog contain subparts identified as “hot” regions that can be selected for examination or order via either pen or speech input.

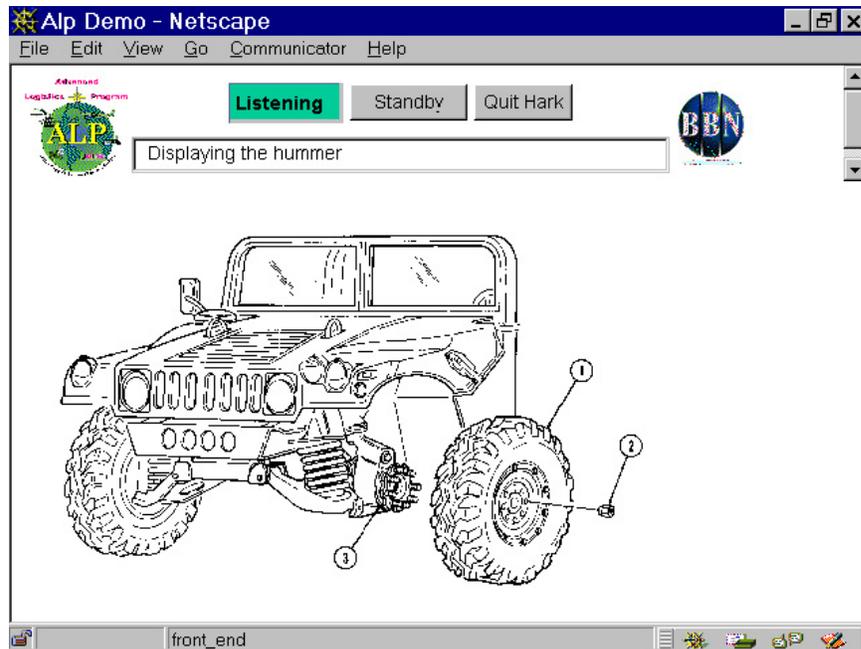


Figure 2.8: VoiceLog Interface

The following is an example usage scenario for VoiceLog (U stands for user input; VL stands for VoiceLog response):

- U:** “Show me the humvee.” [humvee is a nickname for HMMWV military jeep]
- VL:** displays an image of the HMMWV.
- U:** “Expand the engine.”
- VL:** flashes the area of the engine and replaces the image with a part diagram for the engine.
- U:** (while pointing at a region with the pen) “Expand this area.”
- VL:** highlights the selected area, the fuel pump, and displays an expanded view.
- U:** (points to a specific screw in the diagram) “I need 10.”
- VL:** brings up an order form filled out with the part name, part number and

quantity fields for the item.

Similarly, the order form can be filled out with multimodal input by using the pen to select a form fields and speech or written pen input to specify the value. Unique features of the Portable Voice Assistant include its modular multimodal architecture designed for reuse, a centralized speech recognition server, a simple single windowed interface, and a Web-based architecture. Research into the PVA influenced EMMET's modular architecture for supporting extensibility and reuse. The EMMET API is also especially designed for developing Web-distributable applications.

## **2.4 Field Medic Information System**

The Field Medic Information System by NCR illustrates a medical use for multimodal interfaces which integrate speech and pen recognition. This system allows medical personnel to remotely document patient care and status information in the field [23]. This information is then electronically sent to the hospital for patient arrival preparation. Hardware used for the Field Medic system consists of a small wearable computer and attached headset with microphone and earphones called the Field Medic Assistant (FMA), and a handheld tablet computer called the Field Medic Coordinator (FMC). As the medic inputs patient data into the FMA via speech, it is confirmed by a ping sound to indicate acceptance or a click sound to indicate a problem. Also, through speech commands, a medic can request that past input be read back. The FMC tablet computer displays editable medical information about the patient and is wirelessly connected to the FMA such that input and patient data is shared. This connection allows the patient record to be updated and annotated through FMA speech input as well as through stylus input directly to the tablet. Cooperation between the devices

also allows the medic to select areas of the patient record to annotate on the FMC and specify the content of the annotation with speech. Note that unlike some of the pen and speech based interfaces discussed earlier, the Field Medic input modalities are not designed to be used simultaneously. The usefulness of the multimodal Field Medic Information system lies in its hands-free speech input interface coupled with the ability to input more complete information with the tablet interface as needed.

### **3 Multimodal Interface Usability Studies**

The advantages discussed in the prior section provide an impetus to design and research the usability of multimodal interfaces. The benefits of answering questions pertaining to how and when to use such interfaces are apparent. The following studies by Oviatt[47, 42] and Xiao[61] are representative of the types of studies for which the EMMET API was developed. These studies research human factors related to the integration and synchronization of multimodal inputs and provide user statistics to guide designers in the development of multimodal interfaces.

#### **3.1 Human Factors in Integration and Synchronization of Input Modes**

Oviatt's study into the integration of input modes during multimodal interaction used a simulated dynamic map system to analyze user interaction with a speech and pen based multimodal interface [47]. The decision to use a map system for testing was based on her earlier work which revealed that users tend to produce more multimodal commands in visual spatial domains [41]. The goals for Oviatt's study were:

- explore the multimodal integration and synchronization patterns that occur during pen and speech based human computer interaction;
- evaluate the linguistic features of spoken multimodal interfaces and how they differ from unimodal speech recognition interfaces;
- determine how spoken and written modes are naturally integrated and synchronized during multimodal input construction.

Participants in the study were asked to perform home selection related tasks for a client using a “Service Transaction System”. The simulated system displayed client information and an overhead map of an area from which a suitable home was to be found. Users could interact with the system through speech, direct pen input on the map, or both.

General results provide insight into user preferences with regard to multimodal interaction. Users possessed a strong preference to interact multimodally versus unimodally during map tasks. 100% used both spoken and pen inputs during the task. User interviews revealed that 95% of users preferred to interact multimodally. The types of individual input constructions occurred with frequencies summarized in Table 2.1:

<i>Individual Input Construction Type</i>	<i>Percentage</i>	<i>Value</i>
Simultaneous Speech and Writing	19%	165
Unimodal Writing	17.50%	152
Unimodal Speech	63.50%	553
Total Study Participants		871

Table 2.1: Types of Individual Input Construction

Command use analysis results implied the existence of three groupings for task action commands, spatial, selection, and general. Spatial commands have a high like-

likelihood of being input multimodally and include graphics input and query commands such as add, move, modify, or calculate physical distance. The second group is selection commands, which have an intermediate chance of being input multimodally and consist of object information query, deletion, and labeling, or zooming in on an object. The third group includes the remaining general commands, which are rarely input multimodally. Usually, these commands are not spatially oriented and do not require a referent object. General commands include, controlling task procedures, scrolling the display, printing, etc. The frequency of each input task action command is summarized in Figure 2.9.

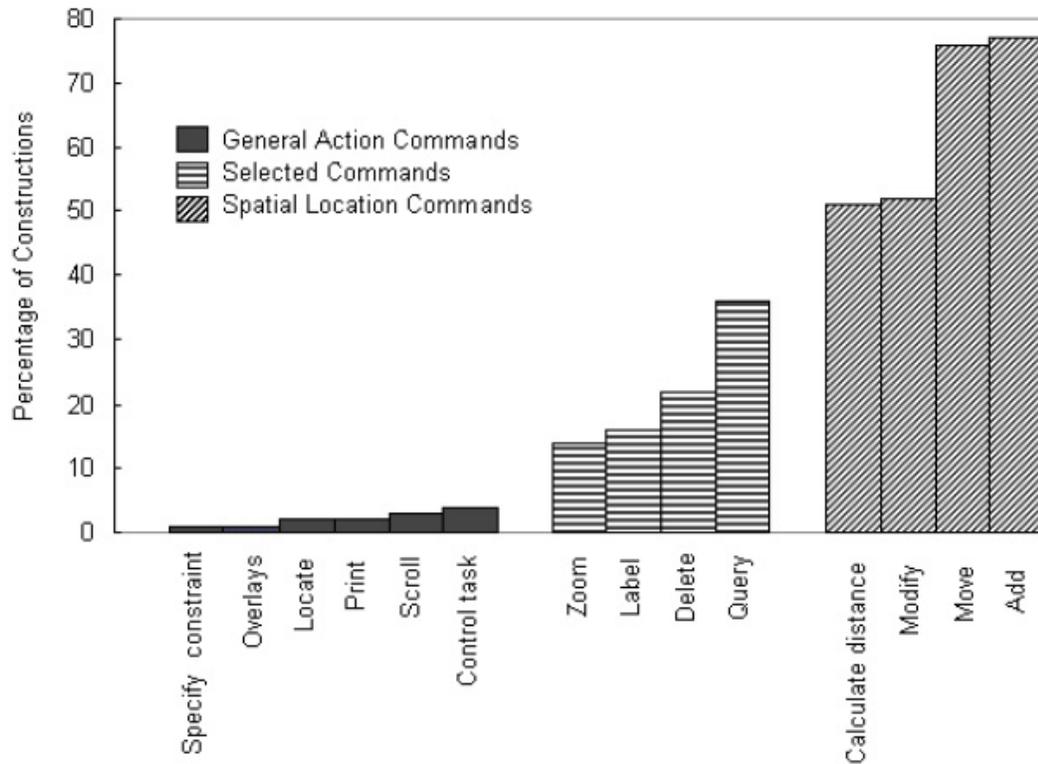


Figure 2.9: Task Action Command Analysis

As expected, the order of spoken semantic elements was based upon typical English subject-object-verb or simply object-verb order. 98% of unimodal speech commands and 97% of multimodal commands adhered to this order. However, a notable result from the analysis of linguistic content was the drastic difference in location of referent objects in command construction. Unimodal speech utterances almost always concluded with the specification of the referent object, rather than indicating the object up front. A unimodal example of this order being, “Add a house next to the dam”, in which the dam is the existing referent object and thus ends the command. In the case of multimodal speech, the results are nearly opposite. When interacting multimodally users overwhelmingly choose to first indicate the referent object or location with the deictic pen gesture, and to then specify the action command. Therefore, an equivalent multimodal command to for the previous example, would begin with the users pointing at the dam, and subsequently uttering, “add house”.

The pen was used in 100% of multimodal inputs used to indicate locations and spatial information. The same information provided by the pen input was duplicated in the corresponding speech utterance about 2% of the time. Conversely, subject and verb elements were specified in speech 100% of the time. Object elements were also spoken about 85% of the time and written 15%. Thus speech and pen input was often complementary.

Table 2.2 illustrates the frequency of different types of pen input for both multimodal and unimodal. Types of pen input include graphics, (e.g. rectangles for building and lines for roadways), symbols or signs, (e.g. X’s indicating deletion, arrows indicating movement), simple pointing, and finally words or digits.

Multimodal usage analysis revealed that most input constructions, 82%, involved draw and speak input, while point and speak inputs represented only 14% of construc-

	Multimodal Constructions by Command Type			Unimodal Pen Constructions
	General Action	Selection	Spatial Location	
Graphic	0%	9.5%	53%	9%
Symbol	25%	33.5%	27.5%	32%
Pointing	0%	57%	14%	0%
Words	75%	0%	5.5%	48%
Digits	0%	0%	0%	11%

Table 2.2: Pen Input Command Type Frequencies

tions. Of the draw and speak constructions 42% were simultaneous, 32% sequential, and 12% were compound (i.e. draw graphic , then point to it while speaking). Sequential draw and speak input constructions were further classified into the nine overlap possibilities. Sequential integrated input constructions were analyzed to determine length of lag between the first input and the second. Integration of deictic terms were also studied to determine when the spoken term and disambiguating pen input were sequential and when they were simultaneous.

Supporting similar multimodal interface usage patterns research was a primary goal in the development of EMMET. The need to provide such support resulted in EMMET’s command and triggers based design. With this design, programmers organize responses to user input into commands (or tasks) and then describe the types and modes of user input that trigger each command. Furthermore, the statistics collection capability of EMMET not only collects general usage statistics, it also reports usage statistics on a per task basis. Thus, innumerable variations upon Oviatt’s Human Factors in Integration and Synchronization study would be facilitated through the use of EMMET.

### 3.2 Mutual Disambiguation of Recognition Errors

A second study by Oviatt contributing to the design theory of multimodal interfaces researches the benefits of mutual disambiguation for avoiding and recovering from recognition errors. An inherent advantage gained from a well designed multimodal interface is the ability to find and correct errors by cooperatively inspecting complementary input modalities for conflicting or confirming content. In Oviatt’s study native and accented English speakers were asked to provide thousands of utterances to be processed by a multimodal system. Participant gender was also a factor. The processing results were logged and analyzed.

The system used for conducting the study was built using the map-based interface and multimodal capabilities of the QuickSet system. Participants were given tasks which involved using the system to set up simulations for community and fire control scenarios. In the process of accomplishing these tasks each subject entered approx. 100 multimodal commands to QuickSet. The research design was a completely-crossed factorial with two between-subjects factors as indicated in Figure 2.10 below:

	<i>Gender</i>	
	Male	Female
<i>Native</i>	Native	Native
<i>Speaker</i>	Male	Female
	Non-Native	Non-Native

Figure 2.10: Mutual Disambiguation Research Design

For the purpose of the study, a mutual disambiguation dependent measure was formulated per subject  $MD_j$ , Eq. (2.1), as the percentage of all their scored commands  $N_j$  in which the rank of the correct lexical choice on the multimodal n-best list  $R^{MM}$  was lower than the average rank of the correct lexical choice on the speech and gesture

n-best lists,  $R_i^S$  and  $R_i^G$ , minus the number of commands in which the rank of the correct choice was higher on the multimodal n-best list than its average rank on the speech and gesture n-best list or: [42]

$$MD_j = \frac{1}{N_j} \sum_{i=1}^N \text{Sign}\left(\frac{R_i^S + R_i^G}{2} - R_i^{MM}\right) \quad (2.1)$$

Other dependent measures analyzed include the ratio of multimodal pull-ups, (situations in which the correct multimodal input was pulled up from a worse-ranked position determined during processing), the percentage of correct speech recognitions and gesture recognitions alone when processing multimodal input, and the total percentage of correct multimodal recognitions.

Results showed that in about one out of eight correctly interpreted multimodal inputs, mutual disambiguation was responsible for the production of the correct response. MD was used more often to correct recognition errors for non-native speakers than native speakers for both male and female participants. Differences in the need for MD based on gender were not significant. These results are summarized in Table 2.3:

<i>Command Length</i>	<i>% Total Commands in Corpus</i>	<i>% Speech Recognition Errors</i>	<i>% MD with Speech Pull-Ups</i>
1 Syllable	40%	58.2%	84.6%
2-7 Syllables	60%	41.8%	15.4%

Table 2.3: MD Rates and Command Length

The ratio of multimodal pull-ups for speech to overall pull-ups was higher for non-native speakers, .65, than native speakers, .35. Successful speech recognition alone was 72.6% for native speakers and 63.1% for non-native speakers. However non-native speakers had a slightly higher gesture recognition rate which contributes to better results in overall multimodal recognition when mutual disambiguation is

performed. The overall multimodal recognition rate remained lower for non-native speakers, 71.7%, than native speakers, 77.2%, however, the difference between the groups was an improvement over the difference in recognition rate for speech alone.

In comparing the success of multimodal input recognition with mutual disambiguation with unimodal speech recognition, result showed an absolute change in command utterance recognition of +13.3%, representing a 41.3% reduction rate in recognition errors.

A secondary goal for EMMET was to support research into other methods of utilizing complementary input modalities. EMMET's ability to define commands with multiple modality triggers, coupled with EMMETS ability to access recorded input events across all modalities, allows further research into the advantages and disadvantages of a wide range of cross modal input evaluation techniques.

### **3.3 Multimodal Interfaces for Children**

Xiao's study into the multimodal integration patterns of children investigates how the multimodal interface interactions of children differ from adult interactions [61]. More specifically the goal of this study was to perform a comprehensive analysis of children's multimodal integration patterns in a speech and pen-based interface and compare the results with integration and synchronization patterns of adults. The analysis of these patterns in children and the comparison to those of adults provides useful data for the development decisions required in the design of multimodal interfaces for children in the future.

Participants in the study were children aged 7 to 10. These children were asked to interact multimodally with a science education application called Immersive Science Education for Elementary kids (I SEE!), which teaches them about marine biology

through animated marine animals. The interface allowed them to interact with these animals by asking questions formulated using speech, pen, or both. They were encouraged to use the input methods as they wish, informed on the instructions where they could write using the pen, and that speech input required that they either tap or begin writing on the screen. They were then left to interact with system for about an hour.

Overall, close to 6500 usable child utterances were obtained. Results showed that speech input alone dominated child utterances, which consisted of 10.1% multimodal input, 80.4% speech input alone, and 9.6% pen input alone. Results also showed that the 93.4% of pen input in multimodal utterances involved abstract scribbling, Of the interpretable pen inputs, consisting of words, symbols, gestures, or pictures, 93.2% complemented the correlating speech input. With regard to user originated error handling, analysis indicated that children were much more likely to use multimodal input when they needed to repair misinterpreted input.

Generally, children preferred simultaneous input over sequential input. Of interest is the discovery that children tended to have a dominant multimodal integration type which they adopted early in the session. When multimodal input was sequential, pen preceded speech in 97% of utterances. Intermodal lag in sequential input involving interpretable pen input averaged 1.1 seconds and ranged from 0 to 2.1 seconds.

In summary, many of the aspects of pen and voice integration patterns are not significantly different in adults versus children. Both adults and children adopt a dominant multimodal input pattern early which is consistently used for later inputs. Also both adults and children used multimodal input to convey complementary semantic information. However, children differ from adults in their likeliness to use simultaneous integration, choosing it in 77% of multimodal inputs versus adult use

in only 36% . Additionally, sequential intermodal lag for children was consistently less for children than adults. Finally, children showed a higher tendency to engage in manual pen activity, if for no purpose other than to scribble.

Xiao's study provides another example of research into multimodal input usage patterns which guided the development of EMMET. EMMET's built-in statistics collection and reporting mechanism was designed to provide task-based usage pattern data analogous to results reported by Xiao, such as percentages of unimodal speech and pen input as well as multimodal input.

## **4 Multimodal Interface Design and Related Work**

### **Conclusion**

The design and implementation of multimodal interfaces is an exciting area of research in the field of human-computer interaction. Initial research in this area includes multimodal systems such as Bolt's "Put-That-There" system which combines speech and gesture, allowing users to identify and act upon referents in speech by physically pointing at their visible representations. Other early systems in this genre include the CUBRICON system, which studies the benefits of maintaining dynamic user and discourse models, to improve interpretation of gesture-based and natural language speech multimodal input, and the XTRA system that also includes the use of user and discourse models while exploring the use of variable granularity in deictic gestures involved in a point-and-speak interface model.

More recent systems include: QuickSet, a reusable map-based speech and pen multimodal interface framework that allows more complex symbol gestures for creating objects as well as spatial and pointing gestures, IBM's Human-Centric Word

Processor, a word processor that explores the benefits of using gesture and speech to edit dictated text, the Portable Voice Assistant, a modular architecture for developing Web-based multimodal applications, and the Field Medic Information System, a portable multimodal system consisting of a wearable hands-free speech interface augmented by a speech and gesture tablet computer interface.

Human factors that need to be considered in the implementation of multimodal speech based interfaces include individual voice quality, short term memory, dialog usage structure, conversational technique, vocabulary, and speech prosody. The human factor of emotion has been the subject of recent studies that explore methods of detecting and addressing emotion during speech input analysis, and designing interfaces that avoid soliciting negative emotions.

Research and implementation of multimodal systems is fueled by the many inherent advantages they provide. Multimodal systems are flexible in their ability to provide users with choice of input. They offer greater availability to a broad range of users. The adaptability of multimodal interfaces is apparent in their capability to switch input modes when situations and environment warrant. The simultaneous input possibilities they provide allow for more efficient input, and the ability of multimodal systems to use mutual disambiguation is an advantage that facilitates error avoidance and recovery.

Finally, usability studies, exploring and evaluating the human factors involved in multimodal input, provide useful insight and guidance toward the design and implementation of multimodal interfaces. These studies include human factors of integration and synchronization of input modes, the use of mutual disambiguation for detecting and corrected recognition errors, and considerations involved in the design of multimodal interfaces for children.

## 5 Multimodal Frameworks Currently in Development

In addition to QuickSet there are a few framework APIs currently in development which are similar to EMMET. The Rutgers University Center for Advanced Information Processing (CAIP) has published their work on A Framework for Rapid Development of Multimodal Interfaces [18], a research group from “AMBIENTE - Workspaces of the Future” has written about their work on A Multimodal Interaction Framework for Pervasive Game Applications, called STARS [31], and a collaborative effort between Penn State University and Advanced Interface Technologies, Inc resulted in a paper on A Real-Time Framework for Natural Multimodal Interaction with Large Screen Displays [62]. There has also been a yet-to-be fully implemented multimodal framework specification for Web interfaces established by the W3C at (<http://www.w3.org/TR/mmi-framework/>).

The Rutgers CAIP rapid development system is a general purpose Java based framework API for multimodal interface development focusing on optimal reusability of the framework code. In addition to this reusability, the system provides a highly generic late fusion multimodal integration agent for managing diverse mode type input sets toward multimodal input resolution. EMMET also addresses this need for highly reusability multimodal development tool. However the CAIP framework focuses on purely the development of multimodal interfaces and the incorporation of new input modalities. EMMET design focuses on the rapid development as well as testing of multimodal interfaces, by providing built-in statistics collection tools for interface design analysis. Secondly, the CAIP framework is limited to the development of multimodal interfaces to 2D applications, whereas the EMMET both supports and

facilitates the development of 3D applications. Finally, EMMET was designed to develop multimodal applications that can be easily distributed and tested over the World Wide Web.

The AMBIENTE STARS system is a .NET based framework supporting multiple input mode types for interactivity with board games and tabletop strategy or role-playing games. It also focuses on support of the social activity inherent in such games via networking. In contrast to EMMET, the the AMBIENTE system restricts it applications to multimodal 2D discrete game environments, whereas EMMET has not such restriction.

The real-time multimodal framework being developed at Pennsylvania State University was designed to facilitate the development of speech and gesture based interfaces involving large screen displays. The philosophy behind Penn State's framework is most similar to that of the EMMET's design philosophy as it is focused on developing an environment for researching certain aspects or problems in multimodal interface design. However, it differs in its focus on questions arising from the use of an output modality such as large screen displays versus EMMET focus on researching multimodal usage patterns.

The aforementioned frameworks represent current published multimodal framework API research efforts. Their relevance to this dissertation is many-fold. They first validate the need for such frameworks as tools to facilitate multimodal interface design research. Secondly, they provide some direction in the methodologies and approaches used for developing such frameworks. However, each of the existing frameworks is constrained by limitations in the types and/or environments upon which its multimodal interfaces are built. Also, none of these frameworks allow for wide dissemination of applications via the Web. Furthermore, none of the framework

APIs in development incorporate an integrated statistics gathering engine to provide the feedback required to facilitate the analysis of the quality of the interfaces built upon them.

# Chapter 3

## EMMET Design and Implementation

### 1 Overview

In undertaking the implementation of a toolkit such as EMMET, programmers must consider the difficulties they may have experienced in adopting new technologies such as programming languages, programming environments and code libraries. To promote the exploration of speech and gesture interface technology, EMMET is designed and implemented to simplify the inclusion of these interface modalities. EMMET is directed at users who are familiar with coding graphics and interfaces in Java, but may have no prior experience or knowledge of how to incorporate speech or gesture recognition into their work. Thus, EMMET's API design strives to dismiss doubts about using speech and gesture, when these doubts arise from the perception that including such modalities is prohibitively complicated or difficult.

The forthcoming discussion of EMMET's design and implementation first pro-

vides background information on the foundational technologies and concepts used to build the API. These technologies include the Java Speech API, HHReco Symbol Recognition Toolkit, jMonkey Engine Java 3D rendering environment, and Multi-modal Integration. Following the discussion of EMMET's foundational technologies, is an in-depth explanation of EMMET's architecture.

## 2 Foundational Concepts and Technologies

The EMMET API is implemented in the Java™ programming language. The decision to use Java is based upon Java's object-oriented features and support for Web accessible content, and upon the ability to use existing unimodal Java speech and gesture recognition APIs. Furthermore, the jMonkey Engine (jME) which is used for rendering the 3D environments is Java based. The speech recognition portion of EMMET builds upon Sun's existing Java Speech API (JSAPI). The JSAPI specification allows Java applications to incorporate speech recognition into their user interfaces. JSAPI's support includes command and control recognizers, dictation systems and speech synthesizers. The implementation of the JSAPI used by EMMET is the CloudGarden TalkingJava SDK. The underlying speech recognition (SR) engine implementation is the engine included with the Microsoft Windows2000 and WindowsXP operating systems. Gesture recognition in EMMET utilizes the HHReco Graphic Symbol Recognition Toolkit [24]. The HHReco toolkit is a Java multi-stroke symbol recognition API developed at UC Berkley by Heloise Hse and A. Richard Newton. EMMET processes input from these APIs in parallel and passes the input events to a multimodal integration agent. This multimodal integration agent is implemented using a time-stamped unification architecture similar to that used by

the QuickSet system [35]. The ultimate capabilities of EMMET include support for creating Web accessible 3D environments as well as support for defining the set of valid multimodal inputs that are appropriate for the particular application utilizing EMMET. Gesture based input support includes recognition of deictic, symbolic, and spatial gestures. Speech recognition input support allows for the specification of a valid set of command utterances.

## 2.1 Java Speech API (JSAPI)

Support for speech recognition in EMMET is provided through the use of the Java Speech Application Programming Interface (JSAPI). The JSAPI is an extension to the Java platform which allows Java applets and applications to incorporate speech recognition into their user interfaces. Its support includes command and control recognizers, dictation systems and speech synthesizers. The Java Speech API also allows Java Web applets and Web accessible applications to access speech capabilities on users' machines. No prior setup of a speech recognition engine is required by the user if Java Web Start is used to dynamically download an implementation of a native speech recognition engine onto the user's machine.

The design of a Java application with speech recognition support using the Java Speech API begins with the definition of the supported rule grammar. This definition is supplied by a Java Speech Grammar Format (JSGF) file. In this file the user defines rules and their associated spoken word set. Figure 3.1 shows a simple example of a JSGF file.

The rules in a JSGF file are enclosed in angle brackets, (i.e. <Command>, <Polite>, etc...). Optional words and rules are enclosed in square brackets. Parentheses can be used to group words and commands. Words, commands, and groupings can be

```
#JSGF v1.0

grammar SimpleCommands; //Define the grammar name

public <Command> = [ <Polite> ] <Actions> <Object> (and <Object>)*;
    <Action> = open | close | delete;
    <Object> = the window | the file;
    <Polite> = please;
```

Figure 3.1: JSGF Sample

followed by an asterisk to indicate that they can occur one or more times.

Vertical bars separates multiple sets of words or actions in which any one of the separated items will satisfy the associated rule.

The static JSGF file is not the limit of what an instantiated JSAPI recognizer can recognize, it is merely a initialization mechanism for forming the base of the recognizer’s valid grammar. In addition to the rules specified in the grammar file, new rules can be dynamically added after the recognizer’s instantiation. In fact, a recognizer can be instantiated with no initial grammar file at all, in which case, the grammar is completely defined dynamically by the application using the recognizer. The ability to dynamically update the grammar recognized is required to allow for the naming of newly created objects in EMMET 3D environments.

The use of the JSAPI for implementing EMMET’s speech recognition support takes advantage of another feature of JSGF called result tags. Result tags are attached to entities in the rule grammar. The syntax used for these tags is overridden by EMMET to store data during multimodal integration. This data is later retrieved to aid in interpreting relevant segments of incoming speech utterances. For example, an excerpt from a JSGF grammar suitable for an interface similar to Bolt’s “put-that-there” system is shown in Figure 3.2.

```
#JSGF v1.0

grammar putThatThereStyle;

public command = <action> <object> [<where>]
    <action> = put {PUT_ACTION} | delete {DEL_ACTION}
    <object> = <pronoun> | ball {BALL_OBJECT} | square
              {SQUARE_OBJECT}
    <where> = <pronoun> | (above | below | to the (left | right)
                        of) <object>
    <pronoun> = (that | this | it ) {OBJECT_REFERENT} | there
              {LOCATION_REFERENT}
```

Figure 3.2: JSGF “Put-That-There” Sample

In this grammar the phrase “put that there” produces three result tags, PUT\_ACTION, OBJECT\_REFERENT and LOCATION\_REFERENT. These results will be time stamped and passed on to the multimodal integration engine to resolve the referent pronouns based on corresponding input from the gesture recognition stream.

The JSAPI speech recognition engine implementation used by EMMET is CloudGarden’s TalkingJava SDK (<http://www.cloudgarden.com/JSAPI/index.html>). The TalkingJava SDK includes full support for the JSAPI specification and Microsoft’s text-to-speech (TTS) and speech recognition (SR) engines. The decision to use the CloudGarden SDK rests on its compliance with the JSAPI standards and its support for academic research via a personal use license of negligible cost. The decision to use Microsoft’s TTS and SR engines is based on the engines’ compliance with the Speech API standard (SAPI) as well as their inclusion with current Microsoft operating systems. Supported speech input devices include built-in laptop or monitor microphones, desktop microphones, and headset microphones. The Microsoft speech recognition engines requires only one brief initial training session, however further training sessions greatly improve speech recognition accuracy.

## 2.2 HHReco Graphic Symbol Recognition Toolkit

EMMET support for symbolic gesture recognition utilizes the HHReco symbol recognition toolkit (<http://www.eecs.berkeley.edu/~hwawen/research/hhreco/>) which is a product of gesture recognition research by Hse and Newton at the Electronics Research Lab, University of California, Berkely [24, 25]. This toolkit provides a Java API to an adaptive multi-stroke symbol recognition system. The toolkit is designed to be used off-the-shelf or customized to suit a specific application. This customizability is used by EMMET to add time-stamp, screen location, and gesture size attributes to recognized input gestures. The gesture classification with added attributes is also used by EMMET for multimodal integration. Devices supported by EMMET for gesture based input include conventional mouse, touchpad, TrackPoint™, and stylus.

## 2.3 The jMonkeyEngine (jME) Java 3D Rendering API

The jMonkey Engine (jME) (<http://www.jmonkeyengine.com/>) is a Java 3D Graphics API. jME provides a scenegraph based architecture for modeling and rendering 3D environments. For the rendering of scenegraphs, jME also provides culling of data to discard scene branches that are not visible to the viewer. The leaf nodes of jME scenegraphs contain the Geometry objects that are rendered to the screen. These geometries can be simple objects, such as cubes, cones, and spheres; objects built from geometric primitives, including Bezier Patches, Lines, and Points; or objects loaded from model files. jME also supports advanced 3D rendering techniques such as environment mapping, terrain generation, lens flare, and particle systems.

The Light Weight Java Game Library (LWJGL) (<http://www.lwjgl.com>) is the high-performance Java 3D rendering engine upon which jME was developed. LWJGL

provides a Java Native Interface (JNI) to the Open Graphic Library (OpenGL), which is the preeminent 3D rendering library supported by most hardware accelerated graphics PC cards.

## 2.4 Unification-based Multimodal Integration

EMMET's multimodal integration agent implementation utilizes semantic unification to produce unimodal and multimodal input interpretations. Unification is basically an operation that takes multiple partial inputs and combines them into a single interpretation. The foundation upon which EMMET's multimodal integration was built was also used to implement the QuickSet system, and is described in a paper by Johnston, et al. on the topic of unification-based multimodal integration [35].

In the unification architecture for Quickset, the unification operation is performed over a typed feature structure. Similarly, the feature structure (FS) for EMMET is consistent for both unimodal speech and gesture input interpretations as well as unified multimodal input interpretations. Figure 3.3 is an example FS for a unimodal symbol gesture. This FS represents the user drawing a cube symbol gesture at  $(x,y,z)$  world coordinates  $(100.0, 0.0, 50.0)$  of proportions resulting in size  $(2.0, 2.0, 2.0)$  at time 10min, 20s, 200ms from application invocation. The interpretation of this FS alone would pass through multimodal integration as a unimodal gesture resulting in the instantiation of a new object of type CUBE with the requested attributes at the requested location.

However, if a temporally relevant speech input FS such as the one shown in Figure 3.4 resulted from the spoken input "Create green cube", then the unification performed by the multimodal integration agent based upon the matching classification, type, and command, would produce the multimodal FS in Figure 3.5. The

```

Gesture_FS
{
  Classification:  OBJECT
  Type:           CUBE
  Command:        INSTANTIATE
  Location:       (100.0, 0.0, 50.0 )
  Size:          ( 2.0, 2.0, 2.0)
  Time:          10:20:200
                :
}

```

Figure 3.3: Gesture Feature Structure

result of this FS would be the instantiation of a cube, similar to the one created by the gesture FS alone, however the cube would also be colored green. Of course the integration agent produces an N-best interpretations list, which would also contain the original unimodal feature structures; both with a lower ranking than the unified multimodal FS.

```

Speech_FS
{
  Classification:  OBJECT
  Type:           CUBE
  Command:        INSTANTIATE
  Color:          'green'
  Time:          10:02:100
                :
}

```

Figure 3.4: Speech Feature Structure

The *N*-best ranking is useful when unification must make assumptions when some conflict in input occurs. If the speech input were more descriptive for example, “Create green cube at 100 units across and 100 units back.”, such that the location coordinates in the speech FS do not match those from the gesture FS, there might be multiple multimodal interpretations. The highest ranking multimodal FS might

```
Multimodal_FS
{
  Classification:  OBJECT
  Type:           CUBE
  Command:       INSTANTIATE
  Color:         'green'
  Location:      (100.0, 0.0, 50.0 )
  Size:         ( 2.0, 2.0, 2.0)
  Time:         10:02:100
                :
}

```

Figure 3.5: Generated Multimodal Feature Structure

average the speech and gesture FS location coordinates followed by two lower ranked multimodal FS for each location coordinate on the N-best list.

## 2.5 Statistics Gathering

A final component of the EMMET implementation is a multimodal usage statistics gathering engine. This component takes advantage of the aforementioned unimodal and multimodal feature structures to collect information such as the percentage of each input type's usage, elapsed time between the modalities used in multimodal inputs, and the percentage of correct results in interpreting each input event. Such statistics are also reported for each command definition defined by the programmer. For Web accessible implementations using EMMET, the results acquired by the statistics gathering engine will be reported to the user. For local implementations using EMMET, the statistics are sent to standard output.

### 3 EMMET's Software Architecture

The following discussion provides an exhaustive explanation of EMMET's software architecture implementation. The progression of this explanation follows the order in which the Java packages being discussed were implemented. Each of these Java packages contains all of the Java classes and interfaces that correspond to a particular aspect of EMMET's functionality.

Some of the terminology used throughout this explanation pertains to the Java programming language and object-oriented programming. Appendix A provides a small glossary to help the reader familiarize themselves with this terminology. Also, for clarification, the terms *developer* and *programmer* refer to those who are using the EMMET API to develop applications, whereas the term *user* refers to the person who uses such applications, i.e. the *end-user*.

In addition, the term *class*, is used exclusively when referring to the definition of an object type, and the terms *instance* and *object* are used when referring to the instantiation of a defined object type or *class*. However, the terms *class*, *instance*, and *object* are not always used when such clarification is not necessary. For example, in the statement, "the x field is defined in MyObject," MyObject must be a class, and in the statement, "the value of x in MyObject," MyObject is an instance of the MyObject class.

Also, common Java CamelCaps conventions are used through the explanation. The term CamelCaps is used to describe a naming style in which name word boundaries are capitalized, as in `myVariableForStoringWidgets` or `MyClassThatDefinesWidgets`. The Java CamelCaps naming conventions additionally state that variables and method names begin with a lower-case letter, as in

`myVariable` or `myMethod()`, and classes begin with upper-case letters, as in `MyClass`. Finally, to clarify when a method name is being referred to, method names will always appear with parens appended, as in `myMethod()` or `onMyCallback()`.

Finally, Unified Modeling Language (UML) class diagram objects are used extensively throughout the explanation. These diagrams provide the proverbial “thousand words” that would be required to explain many of the details relevant to the architecture. Appendix B provides references to UML resources and a brief table for interpreting the icons used in the diagrams.

### 3.1 The *content* Package

As previously described in the discussion of multimodal integration, input events for each modality must be recorded in structures with certain common fields. These fields are required for the eventual integration of multimodal user input. As shown in Figure 3.6, the *content* package serves this purpose with:

- the `ModalContent` class which defines the basic common structure for input events, as well as
- `ModalityType`, `ModalitySubType`, `TimeInterval`, and `ConfidendLevel` classes for the common fields in `ModalContent`, and
- the `ModalContentQueue` class for maintaining `ModalContent` instances in a queue based data structure.

#### 3.1.1 `ModalContent`

The `ModalContent` class diagram in Figure 3.7 illustrates the common fields of the `ModalContent` class. As shown in Figure 3.8, this class is extended for each modality

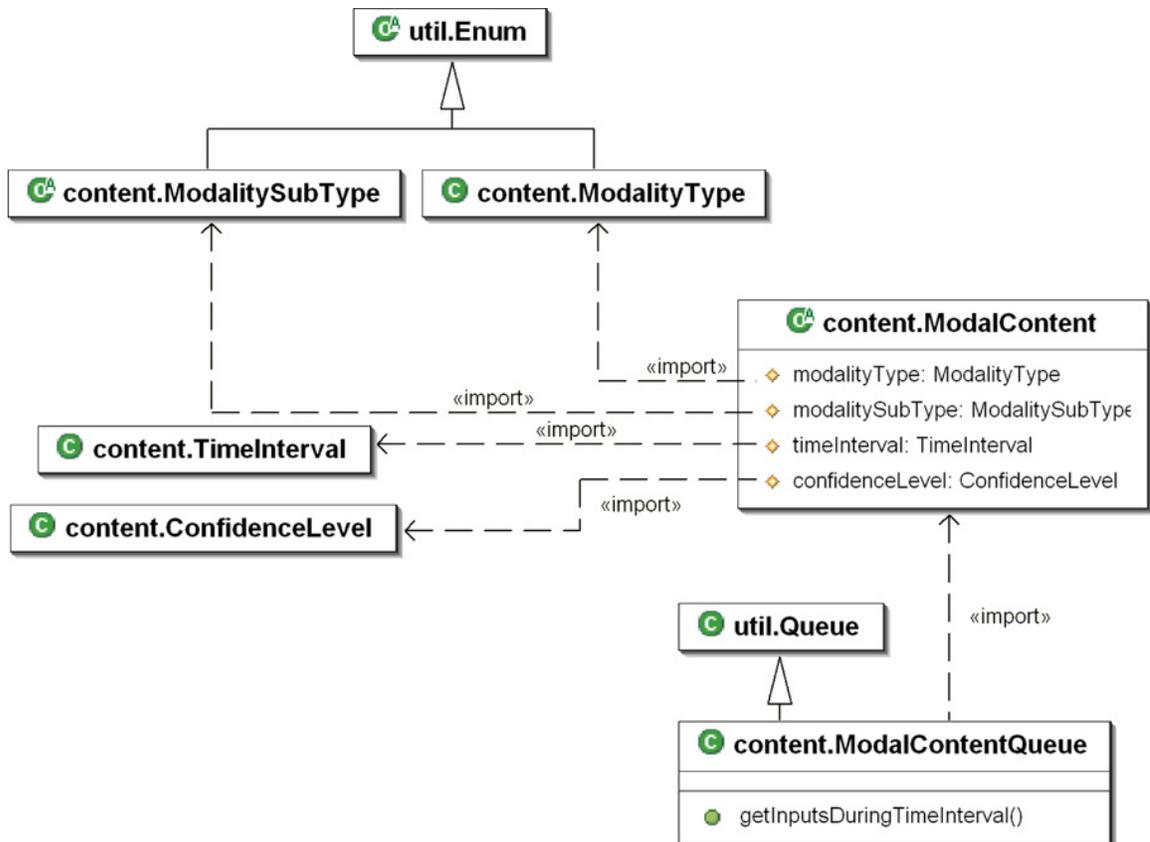


Figure 3.6: *content* Package Overview

to include fields and methods that are specific to events for that modality. During processing of user input, an instance of ModalContent is generated for each discrete input event. This generation occurs in the input manager associated with the input event's modality.

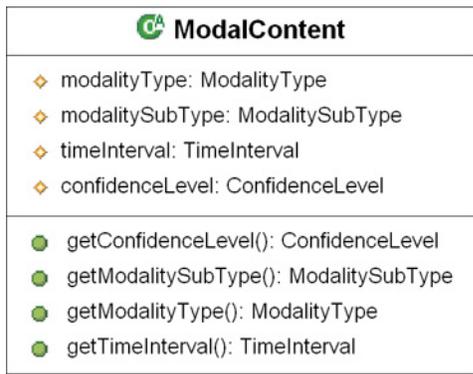


Figure 3.7: ModalContent Class

### 3.1.2 ModalityType

The modalityType field indicates the input modality of a given ModalContent instance. Modality types are instances of the ModalityType class shown in Figure 3.9, which is a strongly typed enumeration of each mode currently supported by the toolkit. The initial toolkit is implemented to support mode types: MOUSE, GESTURE, SPEECH, and KEYBOARD. When an instance of the ModalContent class is generated from multiple input modalities, the ModalityType is set to MULTIMODAL.

The ModalityType class also provides static convenience methods for acquiring a Set or an Iterator of all the current modality types. These convenience methods are often used in code that must respond differently for each type.

When extending the toolkit to include additional modalities, the ModalityType class is one of few places outside of the controller package that needs to be updated.

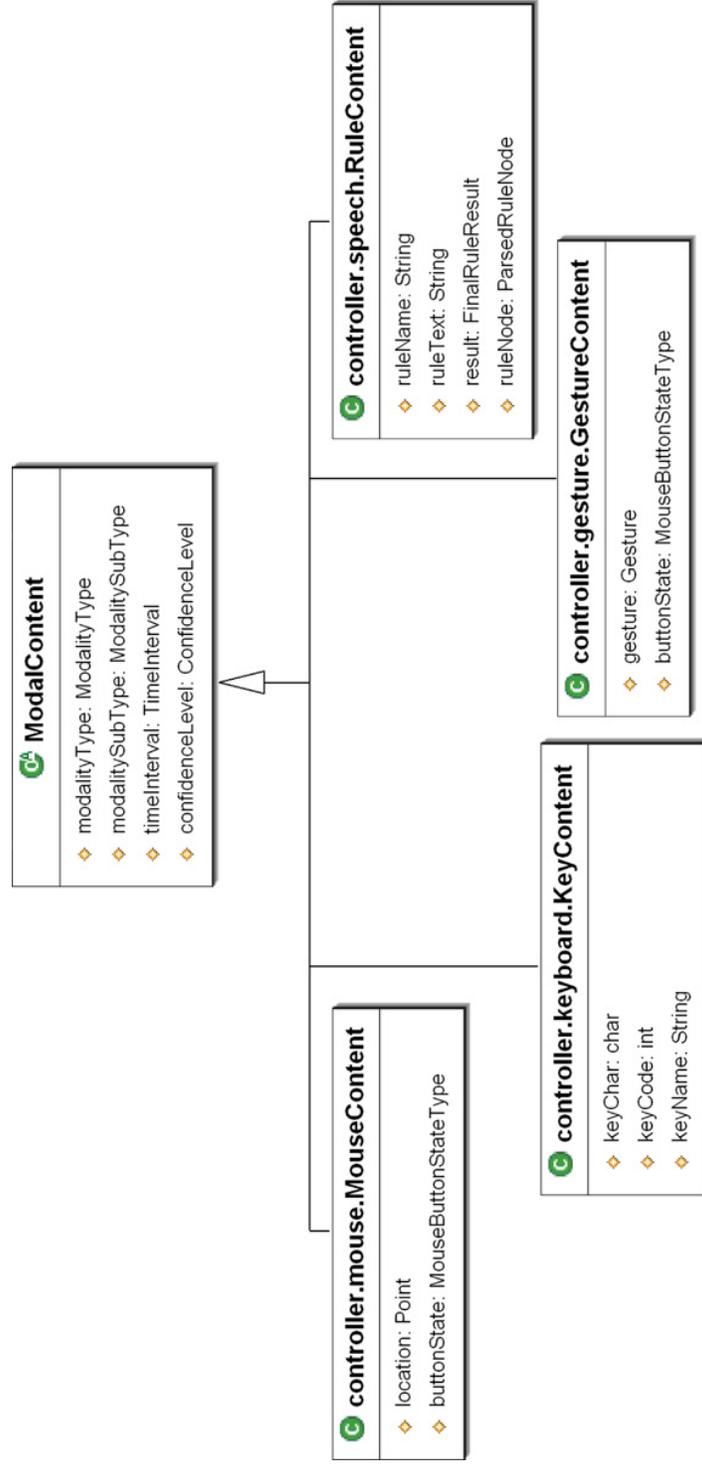


Figure 3.8: ModalContent Subclasses in *controller* Package

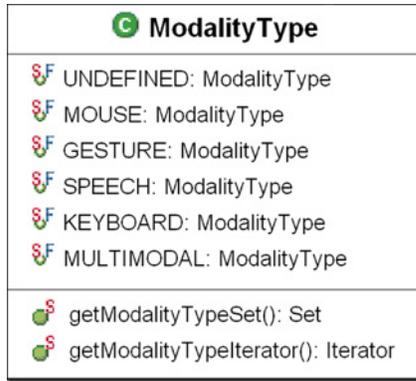


Figure 3.9: ModalityType Class

For instance, the implementation of eye tracking support would require the addition of an `EYE_TRACKING` `ModalityType`.

### 3.1.3 ModalitySubType

The `modalitySubType` field indicates the subtype of a given `ModalContent` instance’s modality type. Modality subtypes are instances of the `ModalitySubType` class, which is a strongly typed enumeration used to provide further details about a specific modality’s input events. Because these details would apply only to that modality, this class is abstract and subclassed for each modality as illustrated by Figure 3.10. For example, the `ModalitySubType` for an instance of `ModalContent` representing a gesture recognition event is defined by the `GestureType` class. The `GestureType` class indicates, among other things, whether the gesture event represented a symbol or spatial gesture.

### 3.1.4 TimeInterval

The `timeInterval` field, defined by the `TimeInterval` class, is critical for the proper integration of multimodal inputs. This field is used during integration to determine

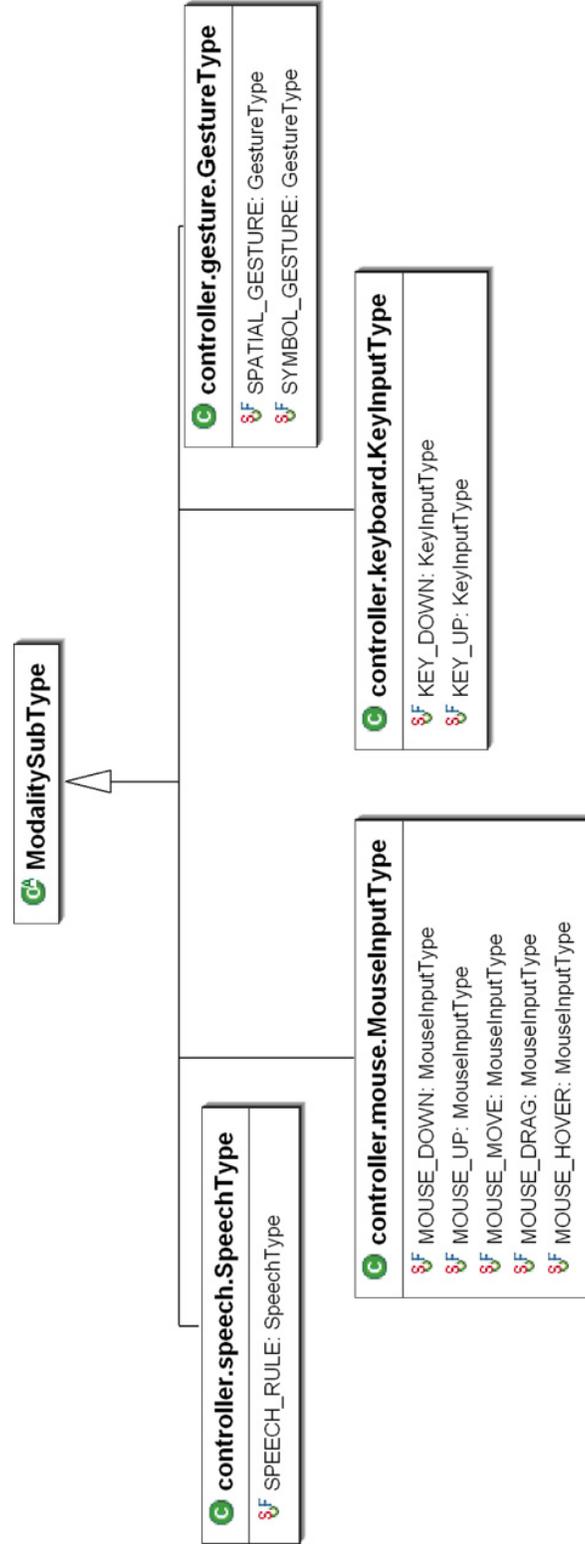


Figure 3.10: ModalitySubType Subclasses in *controller* Package

whether one input event occurred before, during, or after another event. When multiple input events occur during overlapping intervals they may both contribute to the overall interpretation of the user’s input. If one event occurred before or after another, the order of the occurrence may also have an impact on the resulting interpretation.

Figure 3.11 shows the the TimeInterval class which is used to represent the time interval during which a given input event occurred. Within the toolkit, a point in time is consistently represented as the number of milliseconds that have occurred since the initialization of the current toolkit instantiation. During initialization of EMMET, a read-only global field is set to the system clock time in milliseconds. This time can be obtained through build-in Java method, `System.currentTimeMillis()`. For time stamping events, this global field is subtracted from the current system clock time to obtain the current time since initialization.

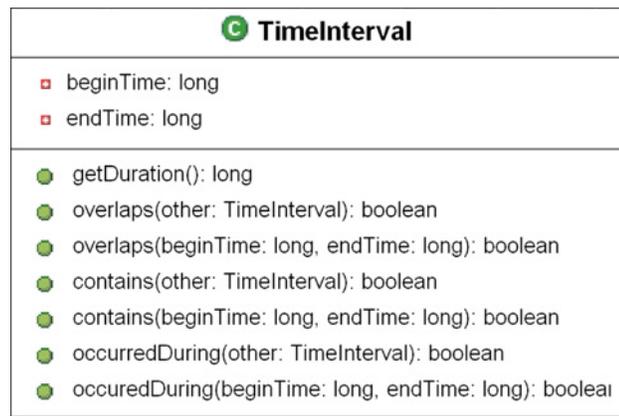


Figure 3.11: TimeInterval Class

Each input event that occurs is represented by a ModalContent instance which is time stamped with a begin and end time. For the purpose of distinguishing between single input events that have no duration versus those that do, the terms *instantaneous event* and *durational event* will be used respectively. For instantaneous events, such

as key presses and mouse clicks, the begin and end time are the same and indicate the time when the actual press or click occurred. For durational events, such as speech utterances and the drawing of symbol gestures, the begin time will be the time when the event initially commenced and the end time will be the time at which the event concluded.

For the sake of determining the number of milliseconds a particular modal input event lasted, the `TimeInterval` class provides a `getDuration()` convenience method. For instantaneous events, it is valid for `getDuration()` to return 0. Also provided by the `TimeInterval` class are a number of convenience methods used for integrating multiple events. These methods answer whether or not a given `TimeInterval` overlaps, contains, or occurred either during another `TimeInterval` or during a given begin and end time.

### **3.1.5 ConfidenceLevel**

The `confidenceLevel` field indicates the degree of confidence with the interpretation of input that led to the generation of a given `ModalContent` instance. Confidence level is maintained by the `ConfidenceLevel` class shown in Figure 3.12 as a float percent value between 0.0f and 1.0f. This confidence is determined by the appropriate input manager at the time input is processed. Certain input events, such as mouse clicks and key presses, have have 100% (or full) confidence. A static convenience method in the `ConfidenceLevel` class, `getFullConfidenceLevel()`, is provided to facilitate the setting a `ModalContent` instance's confidence level to full.

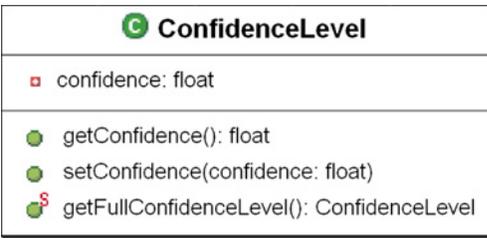


Figure 3.12: ConfidenceLevel Class

### 3.1.6 The ModalContentQueue

To aid in the integration of incoming input events with prior events across multiple modalities, the ModalContentQueue class was implemented. The ModalContentQueue class shown in Figure 3.13 extends a general Queue class implementation and is used to maintain ModalContent instances in the order that their corresponding input events occurred. As ModalContent instances are generated from input events, it is the responsibility of each modality’s InputManager to queue them on the singleton ModalContentQueue instance for that modality. ModalContent instances in a ModalContentQueue may also be removed by an input manager if they expire. The time at which an input event’s generated ModalContent instance expires varies—depending on both the modality of the ModalContent instance and the corresponding InputManager’s adjustable input event expiration time setting. The ModalContentQueue class also extends the Queue class to provide a useful method called `getInputsDuringTimeInterval()` for acquiring a sub-ModalContentQueue containing only the events that occurred between a given begin and end time.

## 3.2 The *controller* Package

The *controller* package consists of subpackages for each modality supported by the toolkit. The implementation of the *controller* package progressed in two stages. In

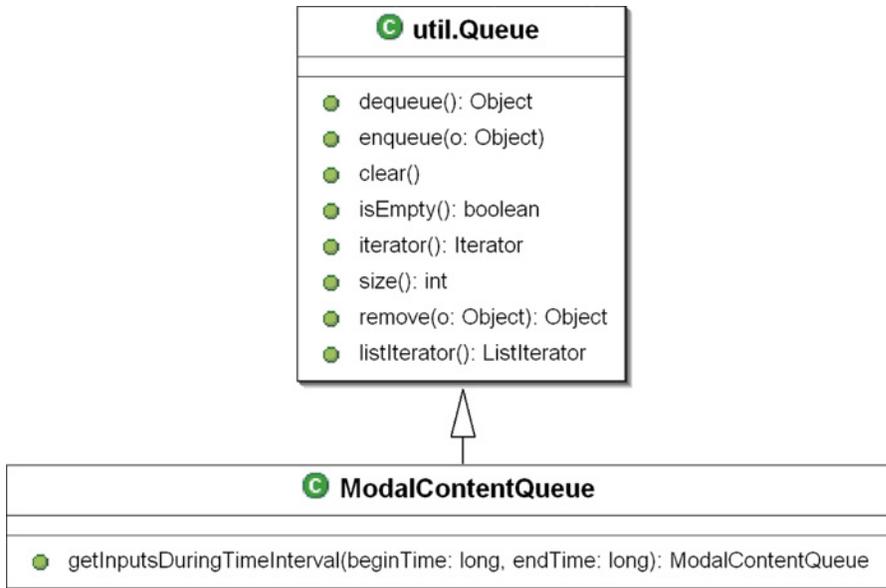


Figure 3.13: ModalContentQueue Class

the first stage the goal was to implement input managers that follow a consistent architecture and provide unimodal support for each modality. In the second stage the goal was full support for multimodal applications through the addition of a *controller.multimodal* subpackage. This subpackage contains a multimodal input manager to instantiate and manage each modal input manager and a multimodal integration agent, suitably named to handle the tasks involved in multimodal integration.

For its respective modality, each controller subpackage contains the implementation of an input manager, an input handler interface with an optional adapter class, a ModalContent subclass, and a ModalitySubType extension. For an example of this architecture see Figure 3.22 in Section 3.2.2 which shows the *controller.gesture* subpackage. This consistency among controller packages supports the goal of an extensible toolkit by establishing the design outline upon which new controller packages

supporting other modalities should be implemented.

### *Input Managers*

Each controller subpackage provides a singleton input manager. One objective for the modal input managers is that they successfully hide any low-level details required to recognize and interpret input events associated with that manager's modality. A secondary objective in the first stage was that each input manager be implemented such that it could be used unimodal

The primary responsibilities for each input manager are:

1. intercepting and interpreting input events for its modality,
2. generating ModalContent instances for each intercepted event,
3. notifying any registered input handlers of each event as it is intercepted, and
4. for the multimodal support stage, queuing generated ModalContent instances onto the manager's singleton ModalContentQueue

The mechanism through which each manager intercepts and interprets input events primarily depends on the modality it supports. Generally, this mechanism involves the manager extending an external library's event listener class then registering itself to listen for events. For instance, to intercept mouse events pertinent to gesture recognition, the GestureInputManager extends jME's MouseInputAction class and registers itself with the jME InputHandler.

Another possible mechanism for intercepting events is direct communication with the associated input device. Each input manager has the option of implementing an `update()` method allowing it to receive control every application update cycle. It is during this period of control, that a manager could communicate directly with an

input device. The `update()` method also provides a time during which the manager can update state information, run tasks, or perform maintenance. The implementation of an `update()` method is optional because it may not be needed by input managers that do not need to perform tasks outside of their input event handling.

Interpretation of input events is also particular to the modality supported. This interpretation must be sufficient to populate the required fields in that modality's `ModalContent` subclass. Some `ModalContent` fields can be set directly from information provided by the input event. Such is the case when setting the location field in a `MouseContent` instance directly from the location value provided in a mouse input event. Other `ModalContent` fields may require the input manager to obtain further information or perform some processing on input events. For example, the gesture field in `GestureContent` is populated with a `Gesture` instance. Upon receiving a gesture complete input event, this `Gesture` instance is constructed by the `GestureInputManager` by interpreting the points recorded from previous input events which occurred while the gesture was being drawn.

Notification of registered input handlers involves iterating through the manager's list of registered handlers and calling their listener callback methods at the appropriate times. The signatures for these callback methods are specified by the input handler's interface. For listener methods called upon the completion of an input interpretation, the object passed is the generated `ModalContent` instance. The objects passed to callbacks listening for in-progress events depend on what information is available at the time. For example, the `GestureHandler` interface provides callbacks that listen for two in-progress events, gesture begin and gesture point added; and one event for gesture complete. The in-progress callbacks only receive location information indicating where the gesture began or where a point was added, while the

gesture complete callback receives the completed `GestureContent` instance generated from the final interpretation of the points collected.

Queuing of generated `ModalContent` instances was implemented as a requirement for multimodal support in the second stage. Each input manager is responsible for maintaining a queue of the input events that have occurred during the interaction session. The `ModalContentQueue` data structure makes performing this queuing task trivial as the task involves simply calling `enqueue()` on an input manager's singleton `ModalContentQueue` instance. The base structure for the items contained in this queue is specified by the `ModalContent` class defined in the *content* package (see 3.1). An input manager may also want to perform additional operations on their `ModalContentQueue`, such as removing queue contents that have been consumed or have expired. The removal of expired inputs is performed when the manager gains control in its `update()` method. The expiration status of a `ModalContent` instance is determined by comparing its `TimeInterval` stamp to the current system time. If consumed inputs are to be removed this removal is done upon their consumption.

All low-level knowledge and implementation details necessary to implement support for each modality is hidden from the other aspects of the toolkit and the programmer by the input managers in the controller subpackages. For instance, the *controller.speech* package encapsulates the instantiation of the speech recognition engine and the rule based grammar; the gesture controller package encapsulates the loading of the gesture file and the recognition of gestures. To extend the toolkit in order to support additional modalities one would be required to follow this design paradigm.

An example of extending the toolkit might be the addition of an eye tracking controller subpackage. To add such a controller one would first need to add an

eye tracking subpackage to the controller package. To remain consistent with the support for existing modalities, this package would need to hide the acquisition and interpretation of video input for the purpose of eye tracking in an eye tracking input manager. To accomplish this task, the input manager may obtain video device input via event listening or polling; some intermediate interface to an eye tracking library; or when it receives control during each system update cycle.

An input manager may also provide methods to support any special capabilities particular to the modality it manages. For instance addition methods are provided by the `SpeechInputManager` to set the recognition mode to either always-on or push-to-talk, and, in push-to-talk mode, to specify what input event toggles the recognition state. A call to some methods may be required to specify the content recognized by an input manager. The `GestureInputManager`, for example, requires the user to call `loadGestureTrainingFile()` to provide a file containing the symbolic gestures it is supposed to recognize.

As previously mentioned, input managers may be used in either multimodal or unimodal input mode. In this context, multimodal input mode entails the simultaneous use of multiple input managers in a manner requiring integration among the modalities each manager supports; whereas unimodal input mode involves the use of each manager separately with no need to cross-reference other modal inputs or access past occurrences of input events.

It is important to understand that the use of the terms multimodal and unimodal in this context applies only to an input manager's current input mode and should not be confused with the concepts of multimodal and unimodal applications. Thus, utilizing the toolkit, a programmer could implement a multimodal application using only managers running in unimodal input mode. Such an application would allow a

user to use multiple modalities, but not in a way that requires the interpretation of combined or accumulated inputs from separate modalities to perform a single task.

To specify and query the current modal input mode, each input manager provides `setMultimodal()` and `isMultimodal()` methods. By default input managers are set to be use multimodally. When multiple managers are used it is required that they all be in the same modal input mode. For further information on how input managers are used and how input events are handled differently based upon the modal input mode setting see section 4: EMMET Proof of Concept Tests and Demonstration Applications.

### ***Input Handlers and Adapters***

Each *controller* subpackage also includes an input handler interface and optional adapter class. The input handler interfaces define callback methods which can be overridden by implementors of the interface. These callback methods fall into two categories. The first category includes methods which are called when a modal input event completes. The parameter passed into these methods is always a `ModalContent` instance reflecting the completed input event. Some input handlers may have multiple callbacks that fall into the first category. For example, the *controller.keyboard* package's `KeyHandler` interface defines `onKeyPressed()` and `onKeyReleased()`. These callbacks both provided `KeyContent` parameters reflecting the content generated from their respective input events. The second category of callback methods encompass notification of any remaining input occurrences related to the supported modality. The optional adapter class provides, often empty, implementations for each callback.

Figure 3.14 shows the *controller.gesture* package's `GestureHandler` interface and `GestureAdapter` class. The `GestureHandler` interface defines the

`onGestureComplete()` callback. This callback falls into the modal input event complete category, thus it receives a `GestureContent` instance generated by the `GestureInputManger` at the completion of a drawn gesture. The remaining `onGestureBegan()` and `onPointAdded()` callbacks fall into the second category, and allow implementors of the interface or extenders of the adapter to receive notification of location information upon the commencement of a new gesture and upon the addition of each point to a gesture.

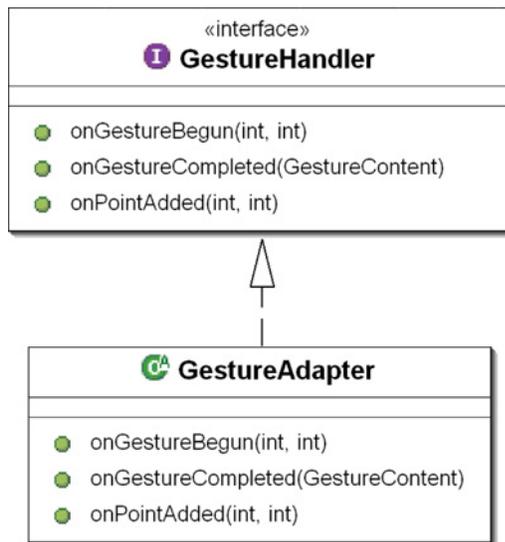


Figure 3.14: Gesture Input Handler and Adapter

An input handler interface must be defined whenever the toolkit is extended to support a new modality. For example, continuing with prior considerations for extending the toolkit to support eye tracking, an `EyeTrackingHandler` would need to be defined. This input handler interface would require at least one callback from the first category. Such a callback might be defined as `onEyeFocusConfirmed()` which would be called upon the completion of an eye focus determined event.

### *ModalContent subclassing*

Each *controller* subpackage also includes a subclass of the ModalContent class. For a description of the base ModalContent class please refer to section 3.1.1. Figure 3.8 shows the ModalContent subclasses for the currently supported modalities. Each subclass extends ModalContent to define additional fields specific to the modality it supports. Each of these fields holds a relevant piece of information that may be derived from the interpretation of an input event. Each ModalContent subclass may also implement query methods. These query methods facilitate the subclass' use by providing convenient access to information:

- additionally maintained by the subclass but not directly reflecting a singular input event aspect;
- extracted from buried or hidden properties of a subclass field whose type is also a class; or
- resulting from calculations or processing performed on one or more subclass fields.

The GestureContent class shown in Figure 3.15 helps to better illustrate ModalContent subclassing. It maintains two protected fields in addition to the ones defined by ModalContent. These fields are `gesture` and `buttonState` which directly reflect the Gesture and MouseButtonState information obtained at the completion of a singular gesture input event. Direct access to these fields is provided by the `getGesture()` and `getButtonType()` getter methods. The query methods `getGesturePoints()`, `getGestureSymbol()`, and `getGestureType()` are all examples of convenience methods that extract information from the GestureContent's `gesture` field. The `getGestureCenterPoint()` query method is an example that returns calculated information. In this case the gesture's center point is calculated

from gesture point data obtained from an internal call to `getGesturePoints()`. Finally, `isSymbolGesture()` is an example of a query method that performs processing on additional information maintained by the subclass and determines whether the current content represents a symbolic or spatial gesture.

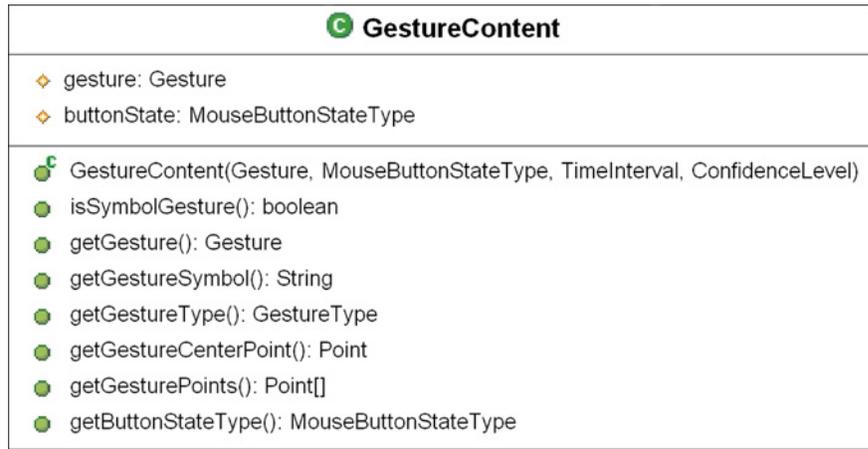


Figure 3.15: Gesture Content subclass

### ***ModalitySubType subclassing***

Finally, each *controller* subpackage provides a subclass of `ModalitySubType`. The `ModalitySubType` base class is described in section 3.1.2. Each `ModalitySubType` subclass provides a strongly typed enumeration of the categories into which that `ModalitySubType`'s modal input events are divided. Figure 3.10 shows the set of `ModalitySubType` subclasses implemented for the modalities currently supported by EMMET. For some modalities, such as speech, all inputs fall into only one category. Thus the `SpeechType` subclass only defines one type, `SPEECH_RULE`. For other modalities there may be a number of sub categories. For example, the `MouseEvent` subclass defines the types: `MOUSE_UP`, `MOUSE_DOWN`, `MOUSE_MOVE`, `MOUSE_DRAG`, AND `MOUSE_HOVER`.

### 3.2.1 The *controller.keyboard* and *controller.mouse* Packages

The LWJGL and jME APIs underlying EMMET provide basic support for handling keyboard and mouse input. LWJGL allows one to query current state information for these devices, such as key pressed state, mouse location, and mouse button state. However, programs that utilize this state query support are responsible for constantly checking device state information in order to perform responsively. jME's input action support builds upon LWJGL's state query support and is slightly more sophisticated. jME's support involves an architecture that allows one to register input action listeners. Included with these listeners is a `performAction()` callback method that receives control only when a device state has changed. When the `performAction()` callback is called, it is passed an input action event object. This object is populated with state information, obtained through LWJGL, corresponding to the input event that initiated the call to `performAction()`

EMMET's keyboard and mouse packages build upon both LWJGL and jME input handling in order to uphold EMMET's goal of providing a consistent input handling interface across all supported modalities. For the discussion that follows, note that *conventional input support* encompasses support for both keyboard input and conventional mouse input and also note that conventional mouse input includes only changes to mouse location and mouse button state.

The singleton `KeyboardInputManger`, shown in figure 3.16, manages EMMET's support for the keyboard modality. It extends jME's `KeyInputAction` class and registers itself with jME as an input handler. As a registered jME key input handler, the `KeyboardInputManager` uses its overridden implementation of the `performAction()` callback to intercept and interpret key events. To elaborate, jME calls the `KeyboardInputManager`'s `performAction()` method for each keyboard input events and passes

an `InputActionEvent` object, reflecting the keyboard input event, as an argument.

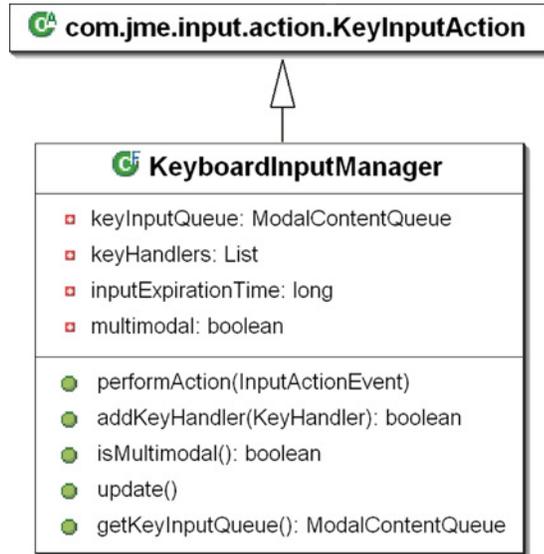


Figure 3.16: KeyboardInputManager Class

As shown in Figure 3.17, the `KeyContent` class extends the `ModalContent` class to include character, key code, and key name fields. These fields respectively hold the Java character, unique LWJGL key identifier code, and Java String representation reflecting a keyboard input event’s key value. An additional piece of information, obtainable from an instance of `KeyContent`, is whether the instance’s corresponding key input event was triggered by a key press or key release. This information is maintained in the `KeyContent`’s `ModalitySubType` field, which contains an instance of the `controller.keyboard` package’s `KeyInputType`. The available `KeyInputTypes` are `KEY_DOWN` and `KEY_UP`. Note that the `KeyContent` constructor performs the task of filling out the base `ModalContent` class fields: `ModalityType`, `TimeInterval`, and `ConfidenceLevel`. For `KeyContent` instances, the `ModalityType` value is `ModalityType.KEYBOARD`; the `TimeInterval` value is a zero length time interval that both begins and ends at the time at which the corresponding keyboard input event oc-

curred; and the ConfidenceLevel is always full (i.e. 100%).

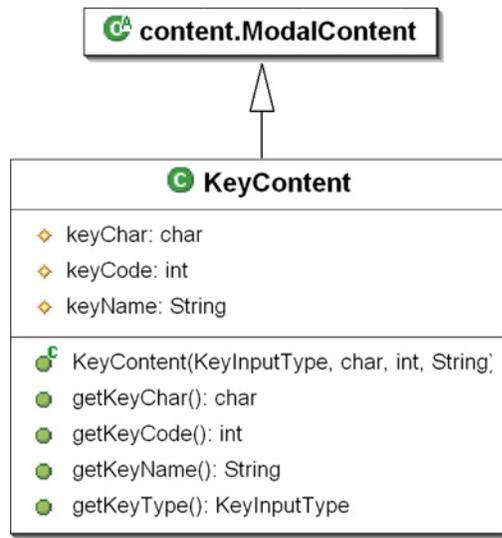


Figure 3.17: KeyContent Class

The KeyboardInputManager’s `performAction()` method generates an instance of `KeyContent`. It populates the fields in this `KeyContent` instance with information obtained by inspecting the `InputActionEvent` argument that `jME` passes into it. The values required to populate the `keyChar`, `keyCode`, and `keyName` fields of a `KeyContent` instance directly correspond to values available through `jME`’s `KeyInput` interface. The `InputActionEvent` class provides access to the `LWJGL` implementation of this interface through its `getKeyKeys()` method. Thus, `performAction()` first calls the `getKeyKeys()` method of the `InputActionEvent` object. The `getKeyKeys()` method returns the associated `KeyInput` instance. The `performAction()` method then queries the `KeyInput` instance for the values it needs to populate the aforementioned fields of the `KeyContent` instance. Finally, `performAction()` queries the state of the key associated with the keyboard input event via the `KeyInput` interface’s static `isKeyDown(int keyCode)` method. In addition to generating a single `KeyContent` instance for each keyboard input event, `performAction()` also supports repeated key event processing

by generating multiple `KeyContent` instances when a key is being held down.

Once `performAction()` has generated the `KeyContent` instance, it notifies all of the `KeyHandlers` registered with the `KeyInputManager`. Implementors of the `KeyHandler` interface register themselves with the singleton `KeyInputManger` by calling the manager's `addKeyHandler()` method. The `addKeyHandler()` method merely adds the `KeyHandler` object it receives to a private list of registered `KeyHandlers` which is maintained by the `KeyInputManager`. The `KeyHandler` interface and corresponding adapter, shown in Figure 3.18, provide two callback methods, `onKeyReleased()` and `onKeyPressed()`. These methods must be implemented or overridden to receive control when the corresponding keyboard input event subtype (i.e. `KEY_UP` or `KEY_DOWN` respectively) occurs. The `performAction()` method iterates through the list of registered `KeyHandlers` and invokes each handler's implementation of the callback corresponding to the current keyboard input event's `KeyInputType` and passes the generated `KeyContent` instance as the only argument.

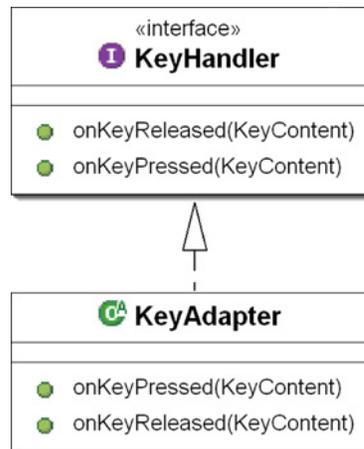


Figure 3.18: `KeyHandler` Interface and `KeyAdapter` Class

Finally `performAction()` determines whether the `KeyboardInputManager` is in multimodal input mode. It does so through an internal call to `isMultimodal()`. If the

KeyboardInputManager is in multimodal input mode, the `performAction()` method performs the additional task of enqueueing the generated `KeyContent` instance onto the `KeyboardInputManager`'s singleton `ModalContentQueue` instance.

The `KeyboardInputManger` also implements the optional `update()` callback. During `update()`, the manager updates the internal keyboard state it maintains for proper handling of key releases, and removes expired `KeyContent` instances from the `key-InputQueue`. `KeyContent` instances expire after they have remained on the queue for longer than the number of milliseconds specified by the `KeyInputMangers` private `inputExpirationTime` field.

The singleton `MouseInputManger`, shown in figure 3.19, manages EMMET's support for conventional mouse input. The term *conventional mouse input* denotes mouse input consistent with conventional WIMP based interfaces, such as a mouse button presses, releases, or clicks; mouse movements; or mouse drags. In a similar manner to the `KeyInputManager`, the `MouseInputManager` extends `jME`'s `MouseInputAction` class and registers itself with `jME` as an input handler. As a registered `jME` mouse input handler, the `MouseInputManager` uses its overridden implementation of the `performAction()` callback to intercept and interpret conventional mouse input events.

The `MouseContent` class, shown in Figure 3.20, extends `ModalContent` to include fields specific to the conventional mouse input modality. These fields reflect mouse device state information associated with a mouse input event. The `MouseContent` `mouseButtonState` field is of the type `MouseButtonStateType` which is a strongly enumerate type defined in `jME`. Each static instance of `MouseButtonStateType` represents a three button combinatorial state indicating which of three possible mouse buttons are currently pressed. For example, `MOUSE_BUTTON_1_3` is a static instance

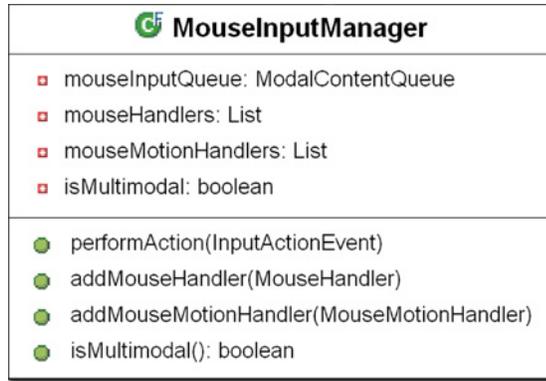


Figure 3.19: MouseInputManager Class

of MouseButtonStateType that represents a mouse state in which the first and third mouse buttons are both currently pressed. A frequently checked MouseButtonStateType is MOUSE\_BUTTON\_NONE which reflects the mouse state where no buttons are pressed. Note that the MouseButtonStateType supports mouse devices with *up to* three buttons. For mouse devices with less than three buttons, MouseButtonStateTypes that represent button state information for the non-existent buttons can be ignored. The MouseContent location field is an AWT<sup>1</sup> Point, which holds the two integer values representing the *x* and *y* location of the current mouse cursor. These *x* and *y* values represent the pixel distance along the *x* and *y* axes of a 2D cartesian coordinate system where location (0, 0) is the lower left corner of the display screen or window.

Also in a manner similar to the KeyInputManager, the MouseInputManager handles the interception and interpretation of conventional mouse input in its `performAction()` callback. All of the mouse device state information needed by `performAction()` to generate an instance of MouseContent can be obtained through the jME MouseInput object. The jME MouseInput object provides query methods for

<sup>1</sup>The Abstract Windowing Toolkit (AWT) is a GUI widget and input device handling library packaged with Java

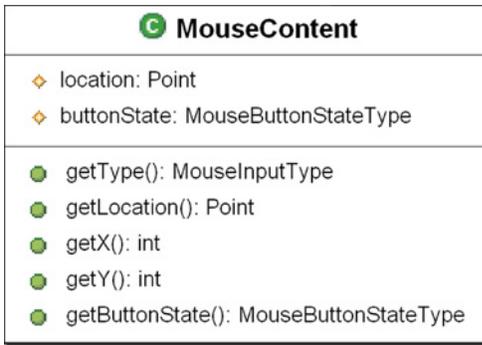


Figure 3.20: MouseContent Class

obtaining information about the current state of the mouse input device. The `MouseListener` maintains a reference to the jME `MouseListener` object for obtaining mouse state information whenever needed. Thus, for each mouse input event, jME calls the `MouseListener`'s `performAction()` callback. The `performAction()` callback then obtains pertinent mouse state information via the handle to the jME `MouseListener` object. Next, `performAction()` uses this mouse state information to generate a `MouseContent` instance that reflects the conventional mouse input event that triggered the call to `performAction()`. Note that the `MouseContent` constructor populates its `ModalContent` superclass fields: `ModalityType`, `TimeInterval`, and `ConfidenceLevel`. Similar to a `KeyContent` instance, an instance of `MouseContent` has a `ModalityType` value of `ModalityType.MOUSE`; a zero length `TimeInterval` value that both begins and ends at the time at which the corresponding mouse input event occurred; and a `ConfidenceLevel` that is always full (i.e. 100%). Upon generating and populating a `MouseContent` object, `performAction()` checks the multimodal input mode of the `MouseListener`. If the manager's input mode is multimodal, `performAction()` queues the generated `MouseObject` onto the `MouseListener`'s `mouseInputQueue`.

Finally, the `MouseListener`'s `performAction()` callback notifies the mouse input handlers that have registered to receive notification of mouse input events from the `MouseListener`. The `controller.mouse` package defines two conventional mouse input handler and adapter sets, shown in Figure 3.21. The first input handler set consists of a `MouseListener` interface and `MouseListenerAdapter` class. This set provides callbacks for mouse button input events only. The second input handler set consists of a `MouseMotionHandler` interface and `MouseMotionAdapter` class. This set provides callbacks for mouse location input events only. Mouse location input events include mouse movement, mouse dragging, and mouse hovering. This separation into categories improves performance by reducing unnecessary calls to handlers that are only interested in one type of mouse input event or the other. `MouseHandlers` and `MouseMotionHandlers` can register with the `MouseListener` by calling the manager's `addMouseListener()` or `addMouseMotionHandler()` methods respectively. The `MouseListener` maintains these registered handlers in `List` objects. This maintenance of registered handlers is similar to the `KeyListener`'s except there are two `List` objects, one for `MouseHandlers` and another for `MouseMotionHandlers`.

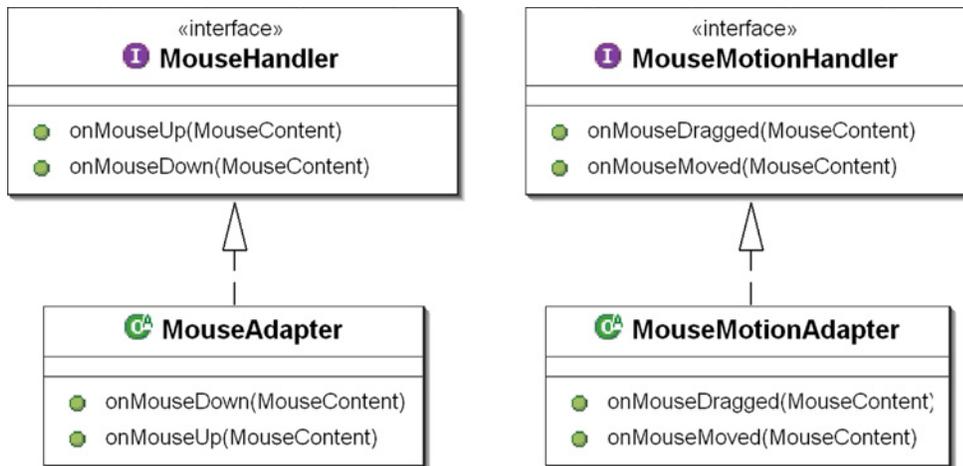


Figure 3.21: Mouse and Mouse Motion Input Handlers and Adapters

### 3.2.2 The *controller.gesture* Package

EMMET's support for symbol gesture recognition is implemented in the *controller.gesture* package. Figure 3.22 illustrates the classes defined in this package and their relationships. As previously mentioned in the discussion of foundational technologies, section 2.2, gesture recognition support for EMMET is built upon the HHReco Graphic Symbol Recognition Toolkit. EMMET uses HHReco to interpret two-dimensional gesture symbols that are drawn on the screen using the mouse cursor. Symbol interpretation involves iterating through an application's set of predefined recognizable symbols to find one that best matches the user drawn symbol.

The singleton `GestureInputManager`, included in Figure 3.22, manages gesture recognition support for EMMET. In the same manner as the `MouseInputManager`, the `GestureInputManager` extends jME's `MouseInputAction` class and registers itself with jME as an input handler. This similarity between the mouse and gesture input managers is a natural result of the mouse being the input device used by both managers. For the purpose of intercepting mouse input events, the `GestureInputManager` implements the `performAction()` method to receive notification of mouse state changes.

The `GestureInputManager` uses an instance of the `HHRecognizer` for recognizing drawn gestures. This recognizer instance must first be trained on the set of symbols it will be used to recognize. The `GestureInputManager` performs this training of the recognizer by loading an HHReco multi-stroke training file. HHReco multi-stroke training files can be created using applications provided in the HHReco toolkit. Once such a file is created, it can be loaded by `GestureInputManager` by calling the manager's `loadGestureTrainingFile()` method. The `loadGestureTrainingFile()` method can load files specified as either a local file or

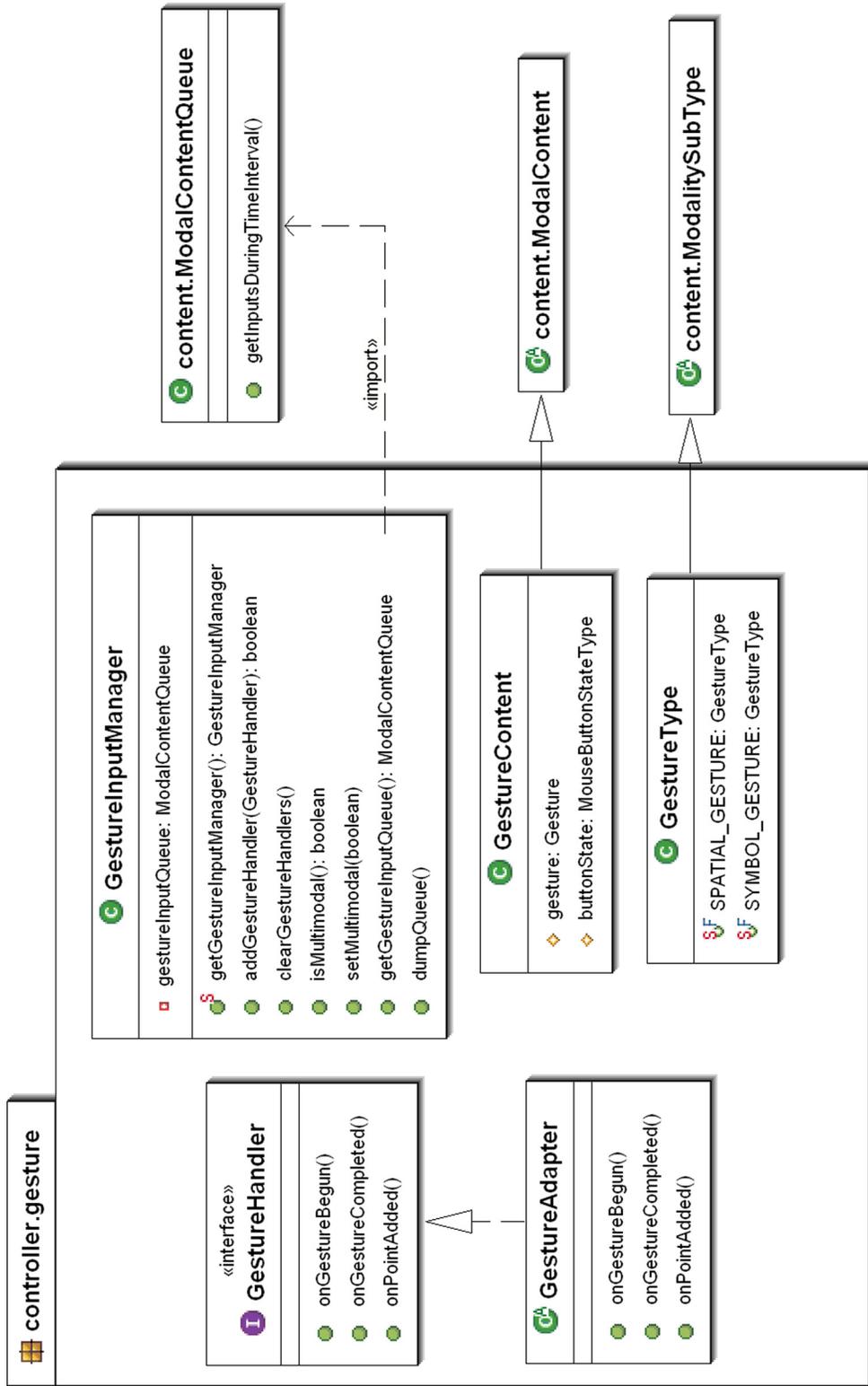


Figure 3.22: *controller.gesture* Package

a URL<sup>2</sup>. The `loadGestureTrainingFile()` method then parses the contents of the training file to extract positive symbol examples which it uses to train the recognizer.

The `GestureInputManager` maintains and updates gesture information in `Gesture` objects. The `Gesture` class, shown in Figure 3.23, uses an instance of the `TimedStroke` class, defined in the `HHReco` toolkit, for storing the gesture points comprising a gesture. The use of `TimedStroke` in this manner necessitates some conversion, because the `TimedStroke` class maintains  $x$  and  $y$  double location values along with a long timestamp value for each point. However, such conversion would have eventually been required as the `HHRecognizer` expects to receive a `TimedStroke` object when asked to perform symbol recognition. The `Gesture` class also maintains the stroke/gesture start time which is used to calculate the timestamps for each point. This start time is provided to the `Gesture` class in a required constructor argument. Other values maintained by a `Gesture` object are an `isGestureComplete` boolean, a symbol name, and a `confidenceLevel`. The `isGestureComplete` value is set when the `Gesture` object is informed that the gesture being drawn has finished via the `setGestureComplete()` method. The `symbolName` and `confidenceLevel` values are set after the recognizer has attempted to classify the `Gesture` symbol. The `Gesture` class also provides convenience methods for retrieving the set of gesture points, the recognized symbol name, and the recognition confidence level, as well as a method that calculates and returns the gesture center point.

While `EMMET` is running, the `GestureInputManager` cycles through three gesture recognition states. This cycle is illustrated in Figure 3.24. In the first state, the manager is waiting for a mouse input event to indicate that the drawing of a symbol gesture has commenced. By default, this indicator is set to be a left mouse button

---

<sup>2</sup>URLs (or Uniform Resource Locators) can specify the path of either a local or remote file

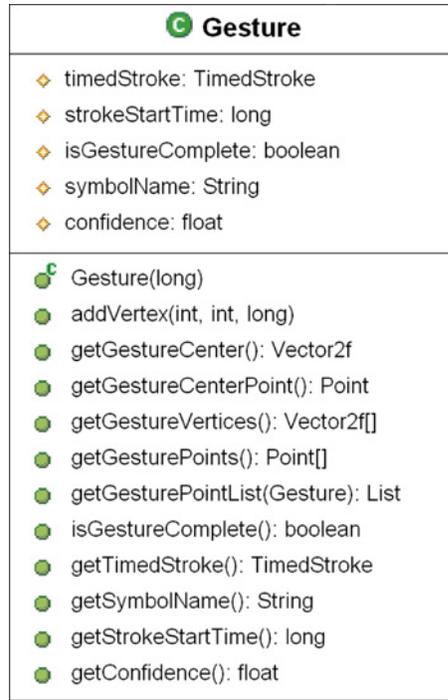


Figure 3.23: Gesture Class

pressed event, however it can be changed to a different one. In the second state, the manager is collecting gesture points and adding them to the current gesture in progress by following the mouse cursor location as the user draws the gesture. The third state is entered when the user indicates that the gesture is complete by releasing the gesture commencement indicator button. In the third state, the GestureInputManger calls upon its HHRrecognizer instance to interpret the drawn symbol. After this interpretation completes, the GestureInputManger returns to the *waiting for gesture commencement* state.

As a registered mouse input handler, the GestureInputManager receives notification of mouse input events through its `performAction()` method. Upon receiving notification of any mouse input event, `performAction()` merely passes control to the `handleGestureInput()` method. The `handleGestureInput()` method is responsible

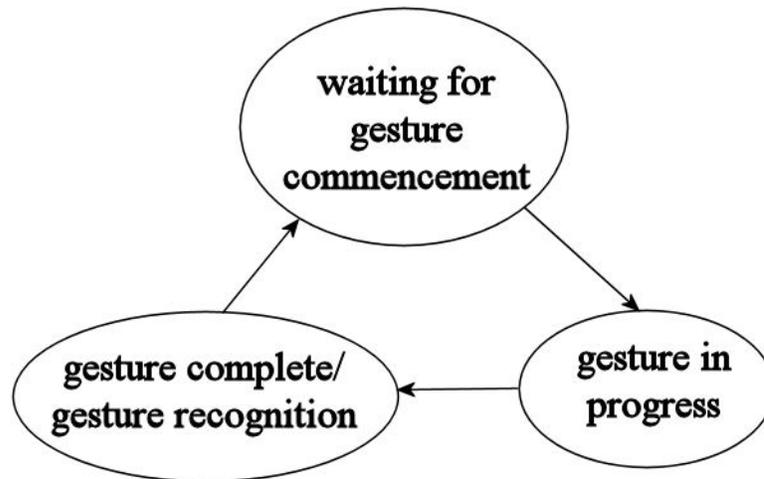


Figure 3.24: GestureInputManager Gesture Recognition State Cycle

for interpreting all mouse input events and responding to those events based on the current gesture recognition state:

- If the manager is in the *waiting for gesture commencement* state, `handleGestureInput()` checks if the gesture commencement indicator mouse button was just pressed. If the indicator button was just pressed, `handleGestureInput()` sets the `GestureInputManger`'s `gestureInProgress` field to a new `Gesture` instance, sets the new `Gesture` instance's first gesture point to the current mouse location, and notifies any registered `GestureHandlers` that a new gesture has begun. The `GestureInputManager`'s `gestureInProgress` field maintains the current `Gesture` being drawn and can be externally referenced via the manager's `getGestureInProgress()` getter method. Upon the commencement of a new gesture symbol, the `GestureInputManger` enters the *gesture in progress* state.

- If the manager is in the *gesture in progress* state and the gesture commencement indicator button is still pressed, `handleGestureInput()` adds a new `Point` to the `gestureInProgress` reflecting the current mouse cursor location. Points are added to the `gestureInProgress` by calling its `addVertex()` method and passing the mouse device's *x* and *y* coordinates along with the current time as arguments. Note that `handleGestureInput()` will only be called via `performAction()`, and that `performAction()` is only called when the mouse device state changes. Thus the `gestureInProgress` should not contain adjacent vertices containing the same location. Upon adding the new point, `handleGestureInput()` notifies any registered `GestureHandlers` that a new point has been added. If the manager is in the *gesture in progress* state and the gesture commencement indicator button has been release, then the `GestureInputManger` enter the *gesture complete/gesture recognition* state.
- If the manager is in the *gesture complete/gesture recognition* state, `handleGestureInput()` first sets the `GestureInputManager`'s `lastRecognizedGesture` field to the completed `Gesture` instance. Next, `handleGestureInput()` obtains the gesture's `TimedStroke` representation from the completed `Gesture` object and passes it to the recognizer. The recognizer then returns a `Recognition` object that contains the recognized symbol name and a recognition confidence level. Finally, `handleGestureInput()` uses the `Recognition` result and completed `Gesture` object to generate a `GestureContent` instance and notifies any registered `ContentHandlers` that a gesture has been completed. The `GestureInputManger` then returns to the *waiting for gesture commencement* state.

The `GestureContent` class, shown in Figure 3.15 , extends `ModalContent` to include fields and methods specific to a gesture complete input event. This class maintains two protected fields, `gesture` and `buttonState`, in addition to the fields defined by `ModalContent`. The `gesture` field contains the completed `Gesture` object used to generate the `ModalContent` instance . The `buttonState` field contains a `MouseButtonStateType` indicating which button was used to draw the gesture. Nearly all of the query methods provided by `GestureContent` are delegate methods for obtaining `Gesture` information from the `gesture` field. These query methods have a one-to-one correspondence with the `Gesture` class' query methods. The `GestureContent` class also provides a getter method for simply obtaining a reference to the `gesture` field.

The *controller.gesture* package also defines the `GestureHandler` interface and `GestureAdapter` class, included in Figure 3.14. This input handler and adapter set allows users of EMMET to listen for gesture input events. The `GestureHandler` interface defines three callback methods for receiving notification of gesture input events. The `onGestureBegun()` and `onGesturePointAdded()` callbacks can be implemented to receive notification when a gesture has commenced or a gesture point has been added to the current gesture in progress. These callbacks are only passed the *x* and *y* mouse cursor location of either the first gesture point or the last point added, respectively. The third callback provided is `onGestureComplete()`, which can be implemented to receive notification of the completion of the last gesture in progress. This third callback is passed the instance of `GestureContent` corresponding to that last completed gesture. The `GestureAdapter` class implements the `GestureHandler` interface to provide empty implementations of each of the three callbacks. `GestureHandlers` can be registered with the `GestureInputManager` by calling the manager's `addGestureHandler()` method.

### 3.2.3 The *controller.speech* Package

The classes and interfaces defined in the *controller.speech* package, shown in Figure 3.25 implement EMMET's speech recognition support. As previously mentioned in the discussion of foundational technologies, section 2.1, speech recognition support for EMMET is built using the JavaSpeech API (JSAPI). The implementations of the JSAPI supported by EMMET are the CloudGarden TalkingJava SDK which utilizes the Microsoft Speech engine, and the IBM Speech for Java SDK which utilizes the IBM ViaVoice engine. The only type of speech recognition supported by EMMET is rule-based speech recognition. Therefore, users specify what speech utterances should be recognized and how a recognized utterance should be interpreted by defining grammars and rules within those grammars. These grammars and rules can be loading from preexisting Java Speech Grammar Format (JSGF) files or specified dynamically by adding new rules using JSGF syntax.

#### ***The SpeechInputManager***

The singleton *SpeechInputManager*, shown in Figure 3.26, manages speech recognition support for EMMET. Unlike the previously discussed input managers, the *SpeechInputManager* does not rely upon jME input handling to receive notification of input events. Rather, it registers itself to receive notification of speech input events from the speech recognition engine. During its construction, the *SpeechInputManager* uses the JSAPI to call upon the speech recognition engine to create a new speech *Recognizer* object. Depending on the currently running speech engine, this *Recognizer* object will be either an IBM ViaVoice or CloudGarden recognizer. The *SpeechInputManager*'s `ENGINE_IMPLEMENTATION` field maintains which speech recognition engine implementation was used. This `ENGINE_IMPLEMENTATION` value

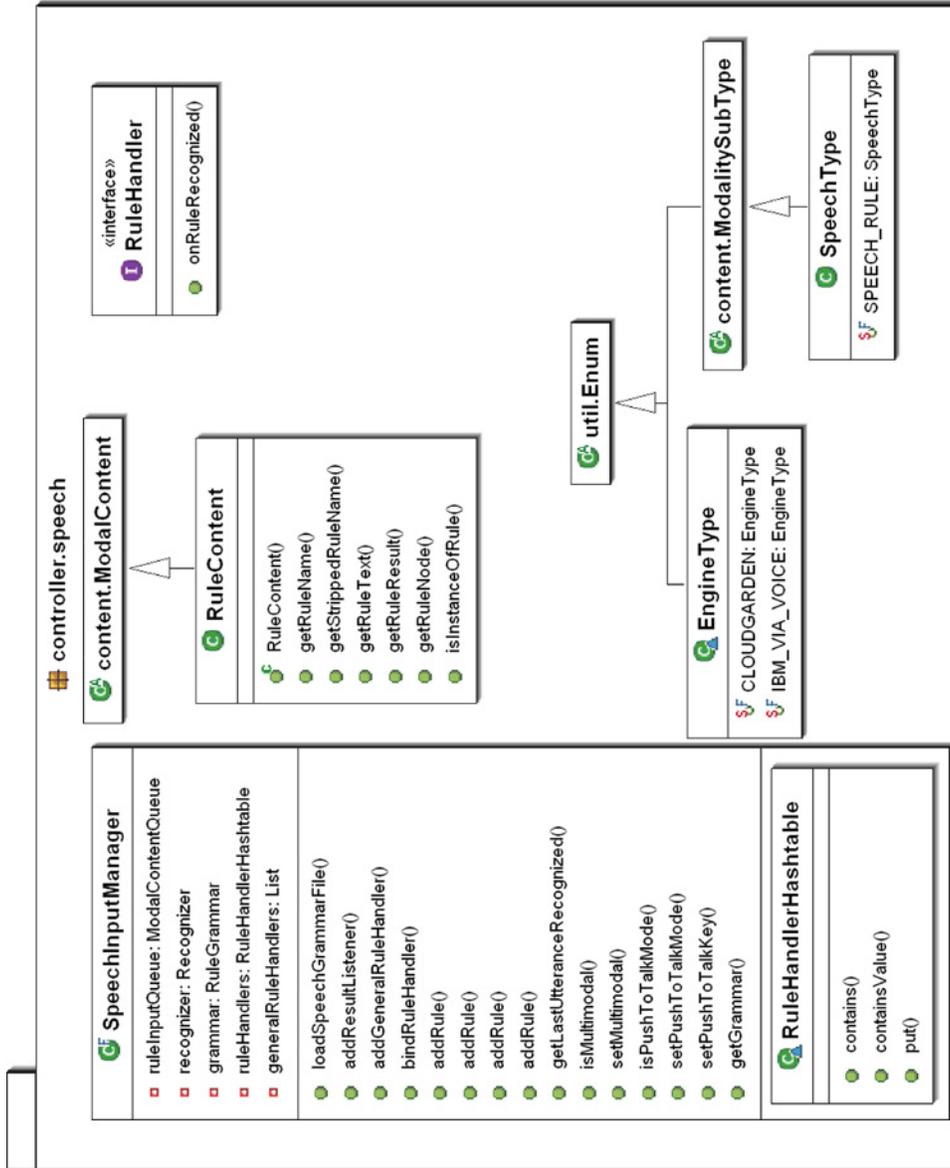


Figure 3.25: The *controller.speech* Package

will be one of two defined `EngineType`'s, `IBM_VIA_VOICE` or `CLOUDGARDEN`. The `SpeechInputManager` references this field to compensate for slight variations in the JSAPI support provided by each engine. Upon obtaining the speech `Recognizer` object, the manger defines a single parent `RuleGrammar` to which all new JSGF rules and sub-`RuleGrammars` will be added. This parent rule grammar is maintained in the manager's grammar field.

Grammars specified in JSGF grammar files can be added to the `SpeechInputManager` by calling the manager's `loadSpeechGrammarFile()` method. New grammar rules can be added to the current grammar using one of the `addRule()` methods. New rules can be specified as a string in JSGF format or as JSAPI Rule object. Added rules may reference previously added rules. Programmers using EMMET may also specify the mode in which speech recognition will occur. In `pushToTalk` mode, the user of an EMMET application is required to hold down a keyboard button to engage speech recognition. When not in `pushToTalk` mode, speech recognition is always being performed. GUI developers using EMMET can specify the speech recognition mode via a call to `setPushToTalkMode()`. The key that engages speech recognition can be specified using the `SpeechInputManger`'s `setPushToTalkKey()` method. Developers also have the ability to manually pause and resume speech recognition via the `pauseRecognition()` and `resumeRecognition()` methods.

### ***Internal Data Structures for Speech Recognition Results***

The `ParsedRuleToken` class shown in Figure 3.27 is used to represent a single word token parsed from a speech utterance. This class maintains a `String` representation of the parsed word token in the `spokenText` field as well as the the time at which the word token began being spoken and the time at which it completed. Figure 3.28

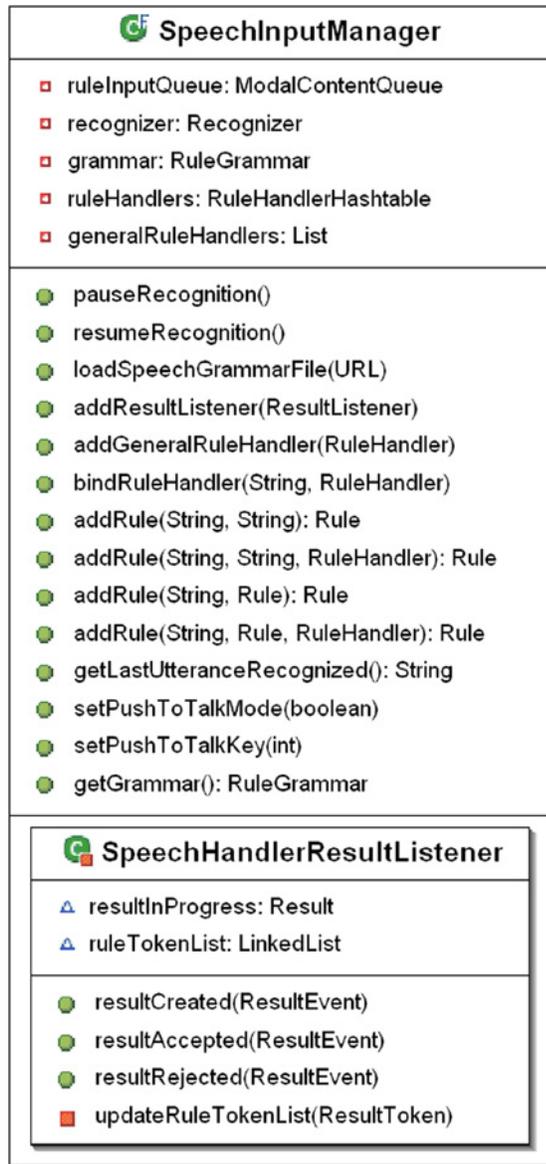


Figure 3.26: SpeechInputManager Class

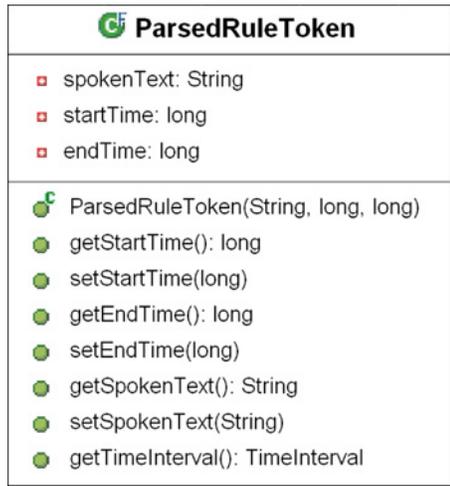


Figure 3.27: ParsedRuleToken Class

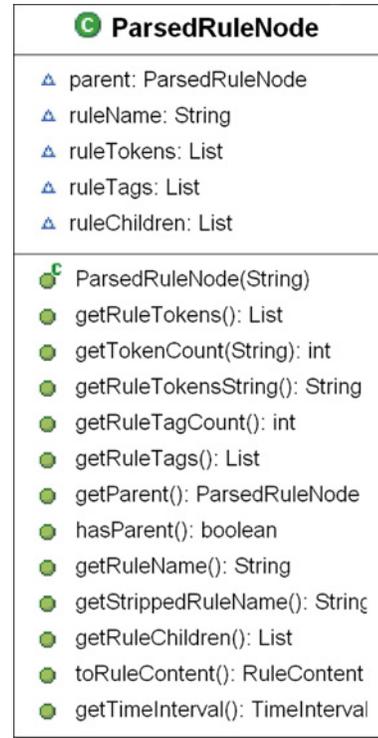


Figure 3.28: ParsedRuleNode Class

shows the ParsedRuleNode. ParsedRuleNodes are used as the basic node type for generating an internal tree representation of a recognized speech utterance. This internal parse tree representation facilitates the interpretation of speech input events for multimodal integration and by users of EMMET.

Given the JSGF grammar in Figure 3.29, Figure 3.30 shows the ParsedRuleNode tree resulting from a recognition of the utterance “Color the green cube red”. The utterance “Color the green cube red” was chosen because of its similarity to phrases used by Bolt to illustrate his pioneering “Put-That-There” multimodal interface. Furthermore, “Color the green cube red” is representative of the types of imperative context sensitive sentences EMMET is capable of resolving.

Note that ParsedRuleNode trees maintain parsed token data cumulatively from

```

#JSGF v1.0

grammar sampleGrammar;

<diectic_ref> = (here | there );
<object>      = [<color>] <object_type>;
<object_ref>  = (the <object>) | ((this | that) [<object>]) | (it);
<color>       = red | blue | green | purple | yellow | orange;
<addObject>   = Add [(a|an)] [<color> {color}] <object_type> {object_type}
                <diectic_ref> {location}
<colorObject> = (color | paint) <object_ref> {target} <color> {color}

```

Figure 3.29: Simple JSGF Grammar for Adding and Coloring Objects

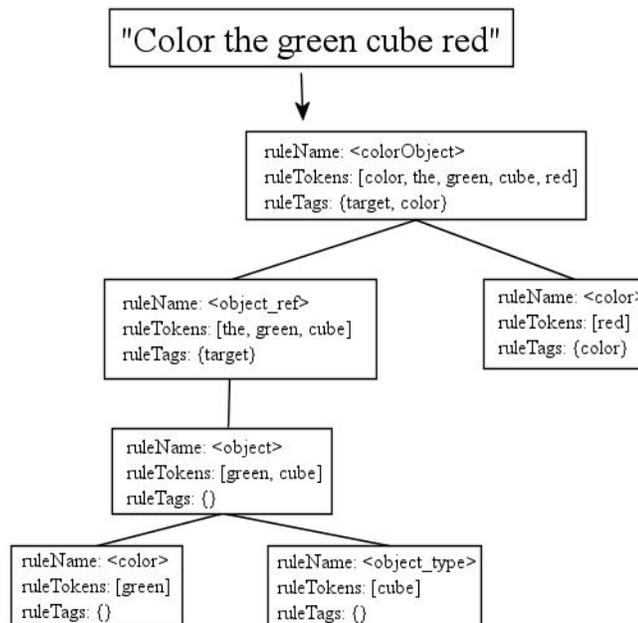


Figure 3.30: ParsedRuleNode Tree

the leaves up through the root. For example, two sibling leaf nodes might each contain one token in their `ruleTokens` list. The `ruleTokens` list for the parent node of these leaf nodes will include the leaf nodes' tokens in addition to the parent's own tokens. The parent node may also have a token tag in its `ruleTags` list. The grandparent of the leaf nodes will cumulatively contain the leaf nodes' tokens, the parent node's tokens and tag, and the tokens contained in any of the parent node's siblings, and so on.

The repetition of parsed rule data in `ParseRuleNode` trees is deliberate. Because of this repetition, each `ParseRuleNode` contains all of the tokens and tags that comprise the parsed rule represent by the node. Thus, methods that utilize `ParseRuleNodes` do not have to search through a node's subtree in order to collect all of that node's parsed rule data. This repetition of data also simplifies the job of detaching and passing `ParsedRuleNode` subtrees.

Also included in the *speech.controller* package is the `ParseRuleUtil` class, shown in Figure 3.31. The `ParseRuleUtil` class provides a number of static utility methods for processing `ParsedRuleNodes` and `ParsedRuleNode` trees. The method provided by `ParseRuleUtil` that is most utilized by EMMET is the `recursiveRuleParse()` method which takes a `RuleParse` object, defined in the JSAPI and provided by the speech recognition engine, and returns an equivalent `ParsedRuleNode` tree. `ParseRuleUtil` also provides query methods for extracting information from `ParseRuleNode` trees, such as a list containing all of the occurrence of a given rule, and a method for querying the number of tags in a given `ParseRuleNode`.

### ***RuleContent***

The `RuleContent` class, shown in Figures 3.32 extends the `ModalContent` class

ParseRuleUtil	
SF	recursiveRuleParse(RuleParse): ParsedRuleNode
SF	recursiveRuleParse(RuleParse, List): ParsedRuleNode
S	getTagCount(ParsedRuleNode): int
S	getAllChildRuleOccurrences(ParsedRuleNode): List
S	getAllChildRuleOccurrences(ParsedRuleNode, int): List
S	getAllRuleOccurrences(ParsedRuleNode): List
S	getAllRuleOccurrences(ParsedRuleNode, int): List
S	getRuleOccurrences(ParsedRuleNode, String): List
S	getRuleOccurrences(ParsedRuleNode, String, int): List

Figure 3.31: ParseRuleUtil Utility Class

with fields and methods specific to the speech recognition modality. As previously mentioned, the speech recognition support provided by EMMET is rule based and, thus a speech recognition event denotes the recognition and acceptance of a single utterance. A speech utterance is accepted only if its word token sequence can be derived from the SpeechInputManager’s current RuleGrammar. When a speech recognition event is accepted, a RuleContent instance is generated. This RuleContent class holds information about the utterance recognized and the utterance’s relationship with the rule it was derived from.

The ruleName and ruleText fields hold the name and spoken text for the rule recognized. The result field maintains the FinalRuleResult object generated by the JSAPI upon recognizing and accepting an utterance. The ruleNode field references the root node of the internal ParsedRuleNode tree representation of the recognized utterance. Given the RuleGrammar specified in Figure 3.29, the spoken phrase “Add a green ball” would be accepted as it can be derived from the rule grammar. The rule name from which this phrase can be derived is <addObject>. Thus the RuleContent instance generated to reflect this recognized utterance would have its ruleName equal

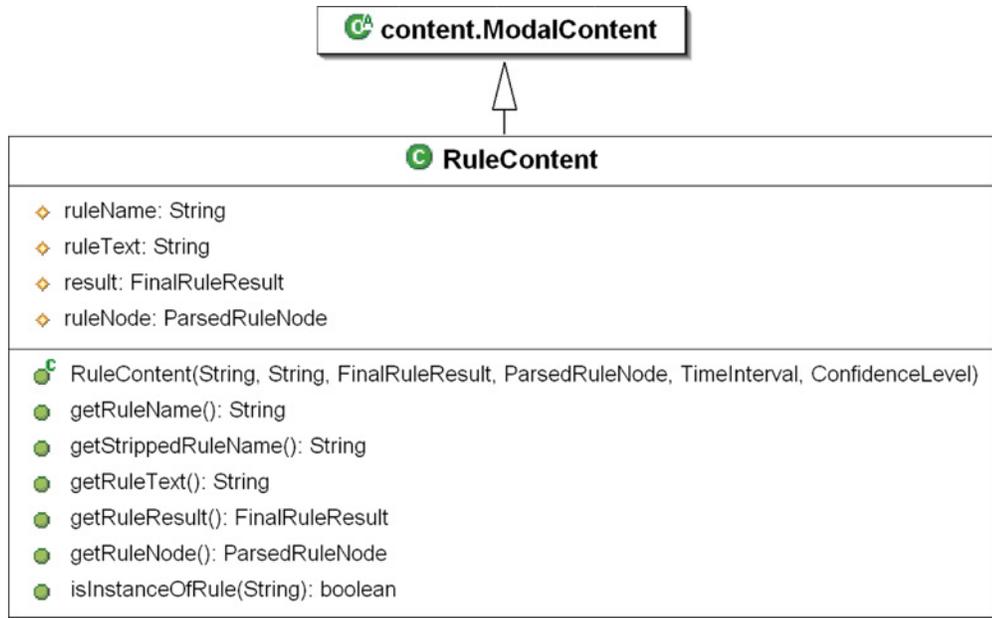


Figure 3.32: RuleContent Class

to “<addObject>” and its ruleText equal to “Add a green ball”. The RuleContent instance’s result and ruleNode fields would respectively hold the utterance’s JSAPI FinalRuleResult and the ParsedRuleNode tree generated from that result.

### ***RuleHandlers***

The additional complexity associated with speech recognition requires some additional complexity in the support for speech recognition event handling. Similar to the input handler implementations for the previously discussed *controller* subpackages, the RuleHandler interface, shown in Figure 3.33, defines the `onRuleRecognized()` callback for responding to speech rule recognition events. Note that no RuleAdapter class was implemented because only one callback is defined in the RuleHandler interface. The adapter classes for the other modalities allow one to override only a subset of the callback methods defined in the implemented handler interface by providing empty implementations of all the handler interface’s callbacks.



Figure 3.33: RuleHandler Interface

The aforementioned additional complexity can be observed in the manner in which RuleHandlers register themselves with the SpeechInputManger to receive rule recognition results. RuleHandlers can register with the SpeechInputManager to receive notification of *any* rule recognition events or they can register to receive notification of only rule recognition events resulting from the derivation of a specified rule or rule set. RuleHandlers registered to receive all speech rule recognition events can also be referred to as *general* RuleHandlers. The way in which general RuleHandlers are registered to receive events is most like the previously discussed handlers. To register a general RuleHandler, the programmer simply passes it in a call to the SpeechInputManger’s `addGeneralRule()` method. Having done so, the general RuleHandler will receive notification of speech rule recognition events in its `onRuleRecognized()`. Information about the rule recognized, including the name of which rule was recognized, can be obtained from the RuleContent instance passed into the callback. However, users of EMMET may need to respond differently to one rule versus another. To address this need, the SpeechInputManger provides a number of ways to request notification of speech recognition events that result from only a specified grammar rule or a subset of rules. Both forms of the manager’s `addRule()` method allow for an optional third argument with which the programmer can specify a RuleHandler object. The RuleHandler specified as this argument is registering to only receive speech recognition events for the grammar rule being adding. A second method for

requesting rule notification only for a specific rule is the `bindRule()` method. The `bindRule()` method takes a rule name and a `RuleHandler` as arguments and registers the `RuleHandler` to only be notified of speech recognition events resulting from that rule.

The `SpeechInputManager` maintains registered `RuleHandlers` in its `generalRuleHandlers` and `ruleHandlers` fields. The `generalRuleHandlers` field is a list structure that holds all general `RuleHandlers` and is treated in the same way as the other input managers' input handler lists. The `ruleHandlers` field is of type `RuleHandlerHashtable`. This `RuleHandlerHashtable` maps each `ruleName` to an associated list of input handlers. This list contains only input handlers registered to receive notification for speech input recognition events that corresponding to the `ruleName`'s grammar rule.

### ***The SpeechHandlerResultListener***

As shown in Figure 3.26, the `SpeechInputManager` also contains an internal `SpeechHandlerResultListener` class. This class extends the JSAPI's `ResultAdapter`. To receive notification of speech recognition input events, a new `SpeechHandlerResultListener` instance is created and registered with the `SpeechInputManager`'s `Recognizer` in the manager's class constructor. State information is maintained by the `SpeechHandlerResultListener` in two fields, `resultInProgress` and `ruleTokenList`. The `SpeechHandlerResultListener` implements a number of callbacks for receiving notification of recognized speech rule events:

- `resultCreated()` is called by the recognition engine upon the creation of a new rule recognition result.
- `resultAccepted()` is called when a rule recognition result is complete and can

be derived from the current RuleGrammar.

- **resultRejected()** is called when a rule recognition result is complete and can *not* be derived from the current RuleGrammar.

When called, the `SpeechHandlerResultListener`'s implementation of **resultCreated()** first sets the listener's `resultInProgress` field to the current speech recognition result in progress. The **resultCreated()** method then initializes the listener's `ruleTokenList` field with a new `LinkedList` instance. The `ruleTokenList` consists of `ParsedRuleTokens` used to store word tokens in the order they are recognized. The `ruleTokenList` is maintained by the listener until the `resultInProgress` completes and the result is either accepted or rejected. Note that a rule-based Recognizer creates a recognition result object immediately upon determining that the word tokens recognized from an utterance in progress could constitute the beginning of a rule grammar derivation. Thus **resultCreated()** is called by the recognizer soon after this determination has been made. Next, **resultCreated()** attaches a new `ResultAdapter` to the current `resultInProgress`. Attaching this `ResultAdapter` allows the listener to be notified when the `resultInProgress` is updated or changed. Upon such notification, the `ruleTokenList` list is updated to properly reflect the `resultInProgress`.

The `SpeechHandlerResultListener`'s implementation of **resultAccepted()** is called when the current result in progress ends and the result can be derived from the `SpeechInputManager`'s `RuleGrammar`. The **resultAccepted()** method finalizes the `ruleTokenList` and passed the finalized list along with the final rule result to the `SpeechInputManager`'s **processRuleRecognition()** method.

The `SpeechHandlerResultListener`'s implementation of **resultRejected()** is

called when the current result in progress ends and the result can *not* be derived from the `SpeechInputManager`'s `RuleGrammar`. The `resultRejected()` method simply sets the `resultInProgress` to be null which, in essence, discards the collected rule tokens and discontinues a speech recognition processing until a new utterance begins.

The final task performed by the `SpeechHandlerResultListener`'s `resultAccepted()` implementation is to call the `SpeechInputManager`'s `processSpeechRecognitionResult()` method. This method processes and interprets recognized speech results created by the speech recognition engine; generates a `RuleContent` instance reflecting those recognition results; and notifies the appropriate `RuleHandlers` and `GeneralRuleHandlers` of the speech rule recognition event.

The `processSpeechRecognitionResult()` method begins by obtaining the recognition result's tokens and tags as well as the recognized rule name from the final rule recognition result. Next, a string representation of the recognized utterance is constructed by concatenating the recognized word tokens. The `SpeechInputManager`'s global `lastUtteranceRecognized` is then updated for later reference. The `processSpeechRecognitionResult()` method next calls upon the current `RuleGrammar`'s parser to obtain a `RuleParse` object corresponding to the recognized text, by passing the constructed spoken text `String` and rule name. The returned `RuleParse` object is used to construct a `ParsedRuleNode` tree representation of the recognized utterance by utilizing the `ParseNodeUtil` utility class. The overall time interval for the recognized utterance is then derived from the start time of the first token and the end time of the last token. Having obtained or derived all of the required information from the recognition result, a `RuleContent` instance is constructed. If the `SpeechInputManager` is in multimodal input mode, the constructed `RuleContent` instance is

enqueued onto the `SpeechInputManager`'s `ruleInputQueue`. Finally any `RuleHandlers` that were registered to be notified when the recognized rule has been recognized are notified, and any `GeneralRuleHandlers` that were registered to receive notification of all speech rules recognized are also notified.

### 3.2.4 The *controller.multimodal* Package

The mouse, keyboard, gesture, and speech *controller* packages discussed thus far were each implemented to provide support for one input modality. The functionality provided by these single modality packages, coupled with the classes defined in the *content* package, is sufficient for users of EMMET to develop applications that only need to utilize the modal input managers in unimodal input mode. Recall that multimodal input mode entails the simultaneous use of multiple input managers in a manner requiring integration among the modalities each manager supports; whereas unimodal input mode involves the use of each manager separately with no need to cross-reference other modal inputs or access past occurrences of input events. In other words, the functionality discussed so far can be used to develop applications that recognize simple speech commands or symbol gestures. However, applications that need to interpret input events from multiple modalities simultaneously in order to reach a single conclusion require multimodal input mode. Applications that need to reference past input in order to properly interpret new input also require multimodal input mode.

The *controller.multimodal* package contains classes and interfaces that contribute to EMMET's multimodal input mode support. It is the one *controller* subpackage that deviates from the design paradigm for implementing *controller* subpackages discussed in section 3.2. The only class in the *controller.multimodal* package class

implemented in accordance with the design paradigm, is the `MultimodalInputManager`. The reason behind this accord is that the `MultimodalInputManager` is the only class utilized outside of the *multimodal* package. The remaining *multimodal* package classes and interface, which include a singleton `MultimodalIntegrationAgent`, a `MultimodalInputListener`, and a `MultimodalInputAdapter`, are only used internally by the `MultimodalInputManger`.

### ***The MultimodalInputManger***

The `MultimodalInputManger`, shown in Figure 3.34, consolidates all of the singleton modal input managers into one multimodal system. Regardless of the multimodal input mode being used, the preferred method for interacting with the singleton modal input managers is through delegate methods provided by `MultimodalInputManager`. The `MultimodalInputManger` provides delegate methods for all of the most commonly used mode specific query any convenience methods provided by each unimodal input manager, as well as delegate methods for adding listeners to each manager. To access modal input managers' less commonly used methods, the `MultimodalInputManager` provides access to each modal input manager through getter methods.

The `MultimodalInputManager` instantiates and initializes the singleton modal input managers independently of how the modal input managers' methods are accessed. During this initialization, the `MultimodalInputManager`'s default behavior is to set the modal input managers to multimodal input mode. The `MultimodalInputManager` is also responsible for calling each modal manager's `update()` method every system cycle.

Some of the requirements for supporting multimodal input mode have already been addressed in the implementation and discussion of the *content* package and

<b>C MultimodalInputManager</b>	
◇	mouseManager: MouseInputManager
◇	keyboardManager: KeyboardInputManager
◇	speechManager: SpeechInputManager
◇	gestureManager: GestureInputManager
◇	integrationAgent: MultimodalIntegrationAgent
◇	statisticsCollector: MultimodalStatisticsCollector
◇	globalStartTime: long
●	getMouseManager(): MouseInputManager
●	getGestureManager(): GestureInputManager
●	getKeyboardManager(): KeyboardInputManager
●	getSpeechManager(): SpeechInputManager
●	bindKeyInputCommand(String, int)
●	loadGestureTrainingFile(String)
●	loadGestureTrainingFile(URL)
●	getLastRecognition(): Recognition
●	getLastRecognizedGesture(): Gesture
●	loadSpeechGrammarFile(URL)
●	addRule(String, Rule): Rule
●	addRule(String, Rule, RuleHandler): Rule
●	addRule(String, String): Rule
●	addRule(String, String, RuleHandler): Rule
●	bindRuleHandler(String, RuleHandler)
●	getLastUtteranceRecognized(): String
●	pauseSpeechRecognition()
●	resumeSpeechRecognition()

Figure 3.34: MultimodalInputManager Class

the modal input managers. One such requirement is the need for a common feature structure for storing input events from each modality. The `ModalContent` class defines such a structure with the common features being the `ModalityType`, `ModalitySubType`, `TimeInterval`, and `ConfidenceLevel`. A second requirement is that input events be recorded somewhere for later retrieval during multimodal integration. This requirement is already fulfilled through the maintenance of a `ModalContentQueue` by each unimodal input manager.

The remaining requirement for supporting multimodal input mode is the ability to perform multimodal integration across all input modalities. This support is provided by the `MultimodalIntegrationAgent`. Like the modal input managers, a singleton instance of the `MultimodalIntegrationAgent` is instantiated and initialized by the `MultimodalInputManager`. In order to process input from all modalities, the Multimodal integration needs to receive notification of all input events from each modality. To register for these notifications, the `MultimodalIntegrationAgent` extends the `MultimodalInputAdapter` class which allows the `MultimodalInputManager` to register the `MultimodalInputAgent` as a handler with each modal input manager. Note that the `MultimodalInputManger` registers the `MultimodalIntegrationAgent` `SpeechInputManager` as general `RuleHandler`.

The `MultimodalIntegrationAgent` extends the `MultimodalInputAdapter` class in order to be registered as an input handler with all of the modal input managers. By implementing each modal input handler's interface, the `MultimodalInputListener`, shown in Figure 3.35, consolidates all of the modal input handlers into one unified listener. The `MultimodalInputAdapter` completes the listener/adaptor pair by providing empty implementations for each method defined in the `MultimodalInputListener`.

A detailed discussion involving the design and implementation of the `Multimodal-`

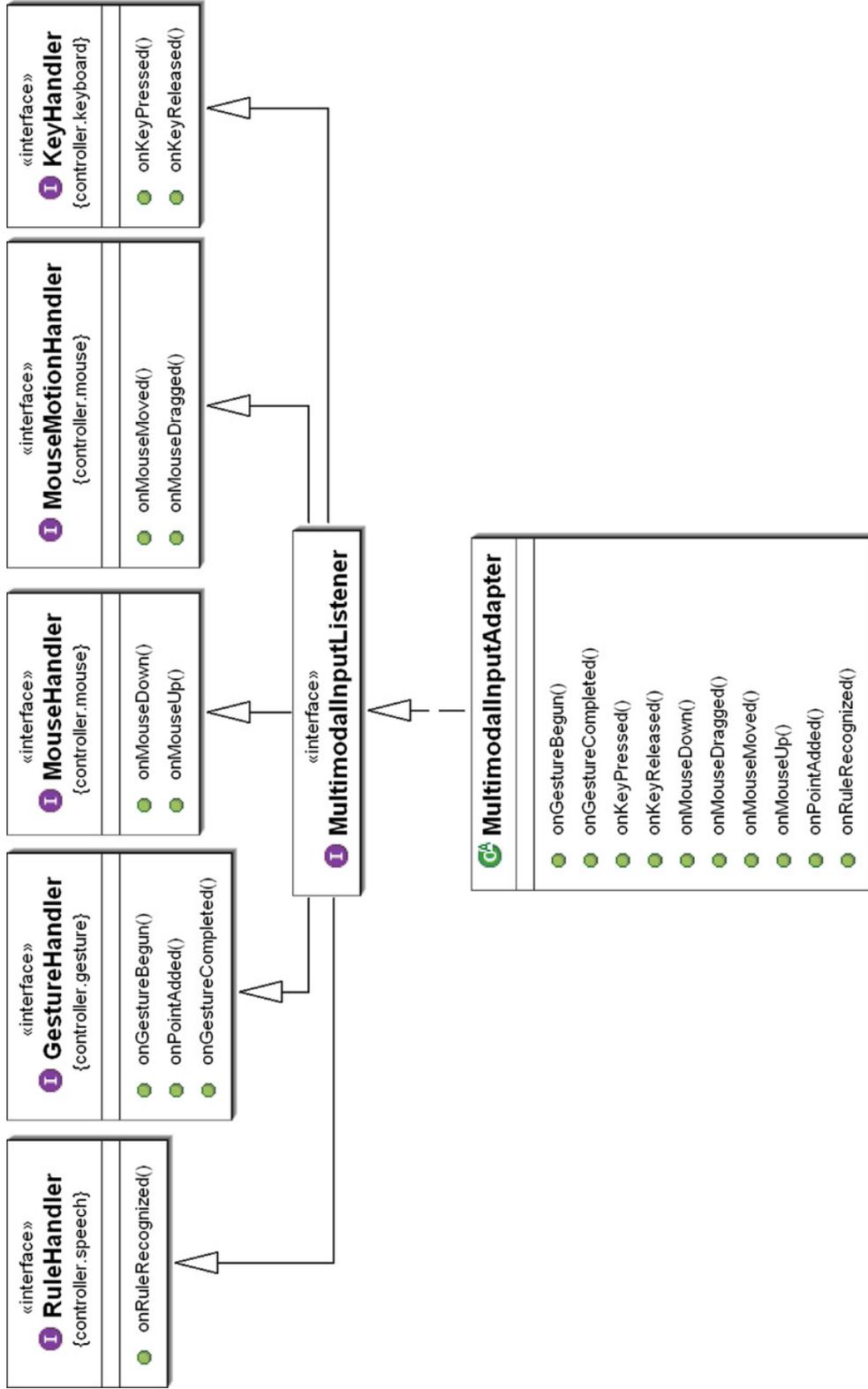


Figure 3.35: Multimodal Input Listener and Adapter

IntegrationAgent itself is greatly facilitated by a clear understanding of the command-and-triggers based paradigm behind EMMET's multimodal input mode support. This paradigm and related concepts are discussed in the next section which covers EMMET's *command* package.

### 3.3 The *command* Package

The *command* package addresses the aspects of EMMET required to support EMMET's command-and-triggers based solution for providing multimodal input mode support. In this solution programmers provide command definitions and specify the combination and types of input events that will trigger the commands defined. The programmer can then register command listeners with these command definitions in order to receive notification when the commands are triggered. This command-and-triggers solution allows the programmer to focus more on responding to the recognition of multimodal input events and focus less on how this recognition occurs.

#### 3.3.1 The *command.trigger* Package

The *command.trigger* package defines the ModeTrigger interface and ModeTrigger implementations for each supported modality. The ModeTrigger interface defines only one method, `isTriggeredBy()`. This method is called internally by the MultimodalIntegrationAgent for determining which triggers are fired by a particular input event. Figure 3.36 shows the ModeTrigger implementor for each modality. Each trigger implementor provides a number of constructors that allow the programmer to describe the type of input events that would fire that trigger.

For example, the SpeechTrigger implementation of the ModeTrigger interface provides three constructors. Two of the SpeechTrigger constructors simply require an

existing speech rule. This speech rule argument specifies which speech rule recognition event should fire the speech trigger. The third constructor defined for the `SpeechTrigger` takes a new speech rule name and a JSFG string defining the new rule. Thus, this constructor allows the programmer to define a new speech rule and create a trigger for that speech rule simultaneously. Once constructed, a `SpeechTrigger` object sets its `ruleName` field to the name of the trigger rule specified in the constructor. As required by the `ModalTrigger` interface, the `SpeechTrigger` also implements the `isTriggeredBy()` method. The `SpeechTrigger` implementation of `isTriggeredBy()` determines if the speech rule event represented by the `RuleContent` object passed into the method matches the rule corresponding to the `SpeechTrigger` object's `ruleName`.

### 3.3.2 The *command.definition* Package

The *command.definition* package contains the classes and interfaces required for creating command definitions and command listeners, as well as a singleton registry for maintaining defined commands.

#### *The CommandDefinition Class*

Instances of the `CommandDefinition` class shown in Figure 3.37 are used to represent user defined commands. The `CommandDefinition` class includes two `List` fields, `triggers` and `commandListeners`. The `triggers` field holds the list of `ModeTriggers` that describes the types and combinations of input events that trigger the defined command. The `commandListeners` field holds the list of `CommandListeners` that have registered to receive notification when the defined command has been triggered. The remaining field is the `commandName` `String`. Pragmatically speaking, the `commandName` should be unique across all registered `CommandDefinition` objects but this

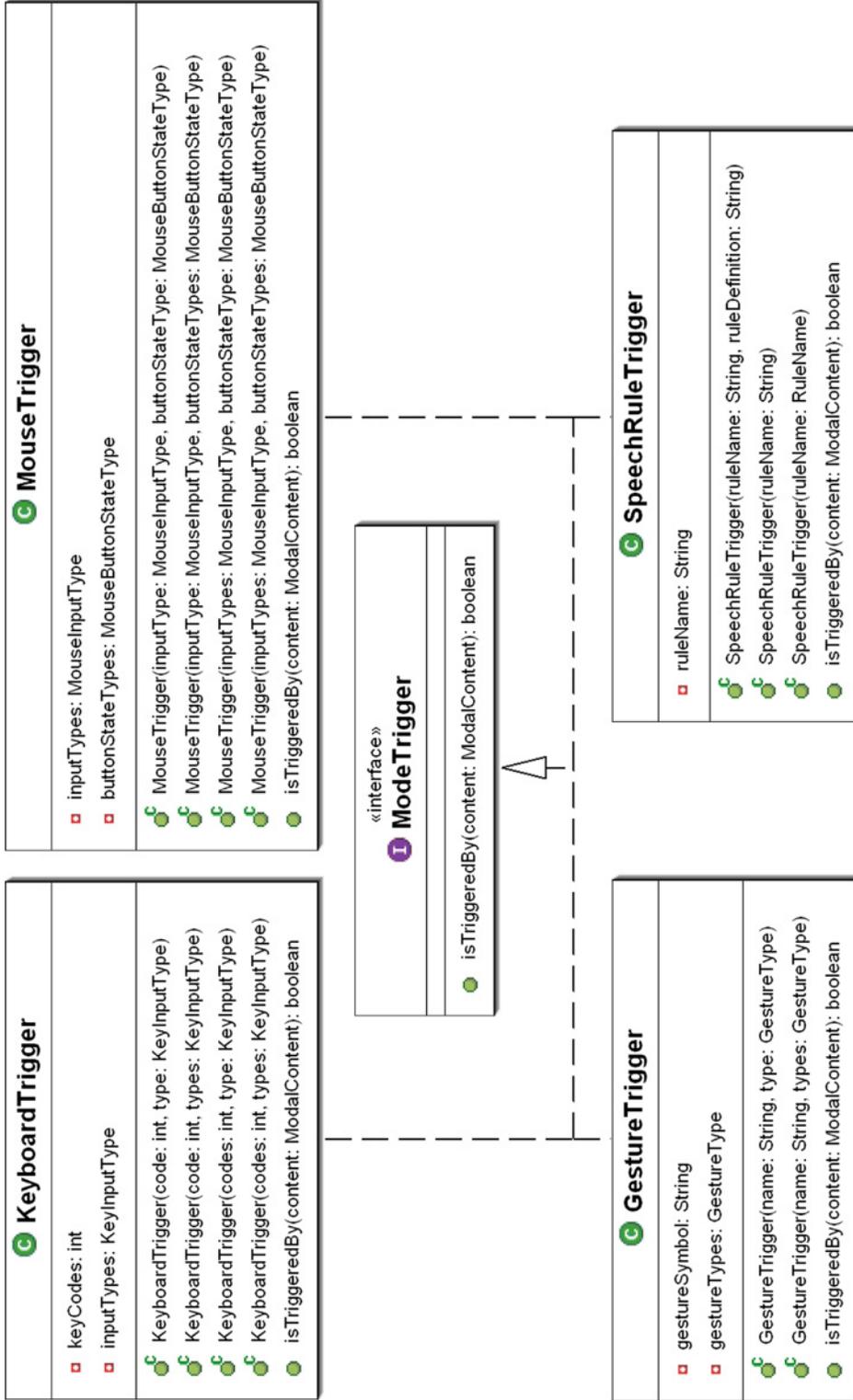


Figure 3.36: ModeTrigger Interface and Implementors

uniqueness is not enforced. The one `CommandDefinition` constructor requires only a `String` argument that specifies the desired command definition name.

The general methodology for manually defining a new command is to first create a uniquely named `CommandDefinition` object via the `CommandDefinition` constructor; then to define and add the `ModalTriggers` for the newly defined command, via the `addTrigger()` method; and to finally register any `CommandListeners` that need to receive notification when the defined command is triggered via the `addCommandListener()` method. Note that the upcoming explanation of the `CommandDefinitionRegistry` will show that this method of manually defining new commands is rarely necessary.

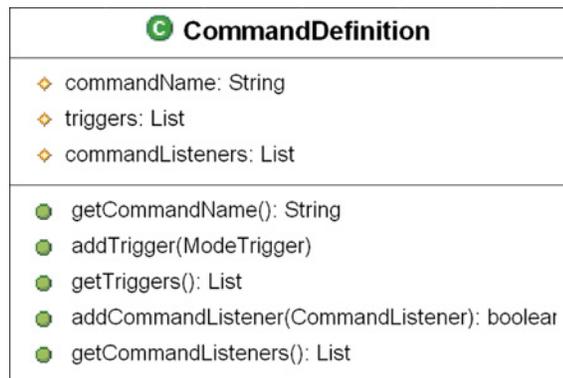


Figure 3.37: `CommandDefinition` Class

### *Command Listeners and Adapters*

As shown in Figure 3.38, the `CommandListener` interface defines callback methods for receiving notification when a command has been triggered. To receive such notifications from a given command, a `CommandListener` object registers itself with that command via that command's `addTrigger()` method. A number of callbacks are defined in `CommandDefinition`. The `onCommandTriggered()` callback is generic

and can be overridden to receive control when *any* of the trigger's command's `ModeTriggers` have been fired. Using the `onCommandTrigger()` callback alone to respond when a command has been triggered most closely follows the paradigm in which the focus is on the command versus the individual modalities. However the `CommandListener` interface also defines mode-specific callbacks for each modality in the form `on<modality>Trigger()` to allow for special handling that depends on the modality of the input event that triggered the command.

The following two scenarios comprise an example that clarifies the use of the general `onCommandTrigger()` callback versus the use of mode specific `on<modality>Trigger()` callbacks. Suppose in the first scenario a programmer wants to define a command for adding cubes to a three-dimensional world. The programmer would like to allow users of her program to add new cubes by drawing a square gesture or saying the phrase "Add a new cube." The programmer would first create a new `CommandDefinition` object with the name "AddCubeCommand." Next, the programmer would create a `GestureTrigger` that is fired when the user makes a square gesture and creates a `SpeechTrigger` that is fired when the user utters the phrase "Add a new cube." The programmer would then add these two triggers to her `AddCubeCommand` object. Next, she would define a `CommandListener` to receive notification when the `AddCubeCommand` is triggered. In this `CommandListener`, she would only need to implement the `onCommandTrigger()` callback because no mode specific information about the input event that triggered the `AddCubeCommand` is needed.

Conversely, suppose in the second scenario that the programmer wants to define a more general "AddObjectCommand". For this command, the programmer wants to allow users of her program to add new objects by drawing various gestures or

by saying the phrase “Add a new \_\_\_\_\_,” in which the user specifies which object to add in the blank. The `GestureTrigger` for this command would need to be fired when the user makes any of the gestures that correspond to createable objects. The `SpeechTrigger` for this command would need to be fired when the user utters the phrase “Add a new `<object_type>`”. However `<object_type>` would have to be defined as a new rule that recognizes words corresponding to createable objects. For this scenario, the programmer would have to implement the `onSpeechTrigger()` callback in order to obtain the word recognized by the `<object_type>` rule. Similarly, the programmer would have to implement the `onGestureTrigger()` callback in order to obtain the gesture that was recognized.

Thus, the mode-specific `CommandListener` callbacks may be required when mode-specific information must be extracted from the triggering input event’s `ModalContents`. However, in the spirit of focusing on the command versus the modalities, programmers can implement one response method that actually handles the command being triggered, and then have all of the mode-specific callbacks call that response method after obtaining any required mode-specific information.

### ***CommandExitState***

The return type for each `CommandListener` callback is a `CommandExitState`. The `CommandExitState` is a strongly typed enumeration. The value returned by an implementation of any `CommandListener` callback must be one of the three `CommandExitStates`, `FAILED`, `IGNORED`, or `SUCCEEDED`.

The `SUCCEEDED` exit state should be returned by a callback implementation when both of the following are true:

1. The input event or events that caused the callback to be notified were indeed

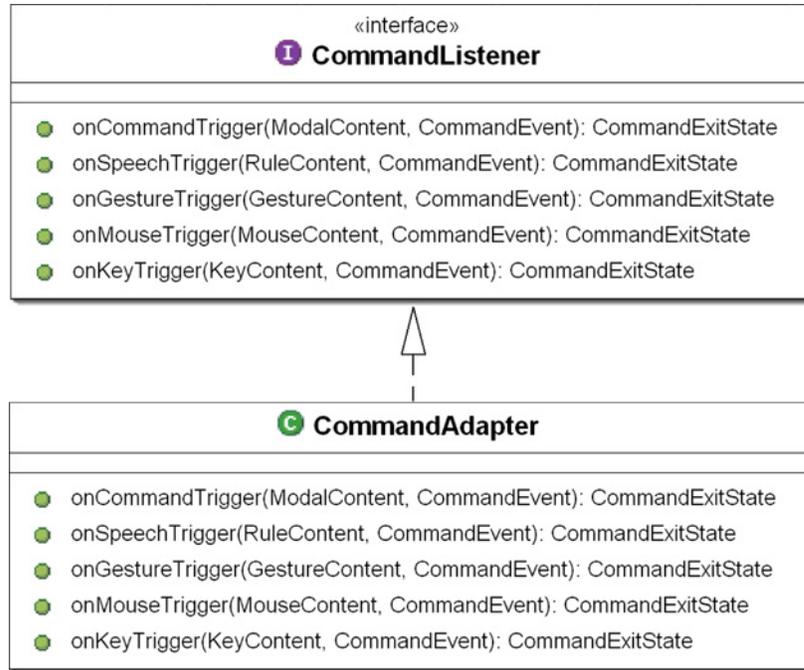


Figure 3.38: The CommandListener Interface and CommandAdapter Class

event(s) meant to trigger the command that the callback is registered with. (Note that it is quite possible for a CommandListener callback to be called for input events meant for a command other than the one the listener is register with if multiple commands have overlapping ModalTriggers.)

2. The callback was successfully able to acquire all information about the input event(s) it needed for its response to the command being triggered.

The FAILED exit state should be returned by a callback implementation when both of the following are true:

1. The input event or events that caused the callback to be notified were indeed event(s) meant to trigger the command that the callback is registered with.
2. The callback was unable to acquire all information about the input event(s) it

needed for its response to the command being triggered.

The `IGNORED` exit state should be returned by a callback implementation when the input event or events that caused the callback to be notified were actually event(s) meant to trigger a command other than the one the callback is registered with.

The `CommandExitStates` returned by `CommandListener` callbacks are used by the statics collection engine for categorizing input events and command events as being successful or unsuccessful. Thus to ensure accurate usage statistics, programmers should make a conscientious effort to ensure that their `CommandListener` callback implementations return the correct `CommandExitStates`. This responsibility of determining whether the notification of a callback was successful is placed on the programmer because it often the case that the programmer is the only entity capable of determining such success.

### ***CommandEvents***

Further observation of the `CommandListener` interface and `CommandAdapter` class shown in shown in Figure 3.38 reveals that all of the `CommandListener` callbacks also receive a `CommandEvent` object. The `CommandEvent` class shown in Figure 3.39 provides an even more complete record of the input event or the combination of input events that triggered a defined command. A `CommandEvent` instance not only contains the `ModalContent` reflecting the input event that triggered the command, it contains a list of all of the `ModalContents` that reflect input events that contributed to the triggering of the command. A `CommandEvent` instance also contains additional information resulting from resolved tags specified in any of the triggering input events. (The use of tags and resolvers will be discussed in detail in the upcoming discussion of the `command.resolvers` package in section 3.4.) Finally, a `CommandE-`

vent instance records the name of the CommandDefinition it triggered allowing it to be passed outside of the CommandListener and adapter system.

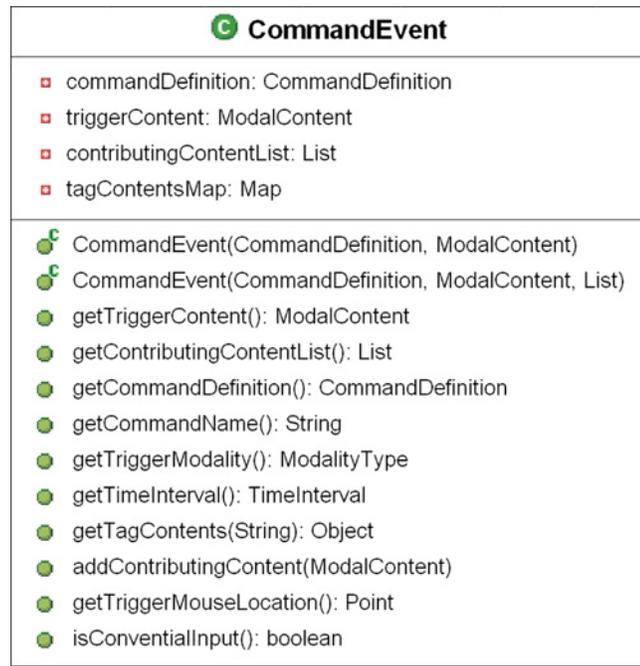


Figure 3.39: The CommandEvent Class

Now that the CommandEvent object has been introduced, note that the triggering ModalContent object is passed to the CommandListener callbacks for convenience as the triggering ModalContent is almost always required for further analysis. If the CommandEvent were the only argument passed to the ModalContent objects, the triggering modality’s ModalContent could be easily obtained through a call to the CommandEvent’s `getTriggerContent()` method.

The ModalContent objects in the contributing content list maintained by an instance of CommandEvent reflect all of the modal input events that contributed to the triggering of the command. For instance, an “addObject” command may be triggered by the user uttering the phrase “Add a ball here” while hovering the mouse

cursor over a location that resolves the diectic “here” reference. In this case, the command was triggered by the “Add a ball here” speech input event and, thus, the `triggerContent` field of the resulting `CommandEvent` object would contain a `RuleContent` object that reflects the speech utterance. However the resolution of the diectic “here” required a mouse input event and, thus, the contributing contents list in the resulting `CommandEvent` object would include a `MouseContent` object that reflect the mouse hover input event. Note, that the discussion of resolvers and tags will reveal how the tag content map in the resulting `CommandEvent` could also contain the screen location of the resolved diectic.

The `CommandEvent` class also provides a number of useful query methods. These query methods include the `getTimeInterval()`, `getTriggerMouseLocation()`, and `isConventionalInput()`, and the self-explanatory `getTriggerModality()` method. The `getTimeInterval()` method returns a time interval constructed by taking the start time of the triggering content and the latest end time from all the contributing modal contents. The `getTriggerMouseLocation()` returns the location of the mouse cursor at the time of the triggering input event. The `isConventionalInput()` method return true if the triggering input event was from the mouse or keyboard.

### ***The CommandDefinitionRegistry***

The singleton `CommandDefinitionRegistry`, shown in Figure 3.40, maintains all active `CommandDefinition` objects. A manually defined `CommandDefinition` is not activated until it has been registered with the `CommandDefinitionRegistry` via the registry’s `registerComandDefinition()` method. However, as stated earlier, the process of manually defining `CommandDefinions` and then manually adding those definitions to the registry is usually unnecessary. This process is unnecessary because

the `CommandDefinitionRegistry` provides a number of convenience methods that allow the programmer to define new `CommandDefinitions` that are then automatically registered with `CommandDefinitionRegistry`.

 <b>CommandDefinitionRegistry</b>	
	<code>commandDefinitions: Set</code>
	<code>addBasicSpeechDefinition(String, String, String): CommandDefinition</code>
	<code>addBasicGestureDefinition(String, String): CommandDefinition</code>
	<code>addBasicMouseDefinition(String, MouseInputType): CommandDefinition</code>
	<code>addBasicMouseDefinition(String, MouseInputType, MouseButtonStateType): CommandDefinition</code>
	<code>addBasicKeyDefinition(String, int, KeyInputType): CommandDefinition</code>
	<code>addBasicDefinition(String, ModeTrigger): CommandDefinition</code>
	<code>addDefinition(String, ModeTrigger): CommandDefinition</code>
	<code>registerCommand(CommandDefinition)</code>
	<code>getCommands(): String</code>
	<code>generateCommandEvents(ModalContent): CommandEvent[]</code>

Figure 3.40: The `CommandDefinitionRegistry` Class

The methods provided in the `CommandDefinitionRegistry` for defining new `CommandDefinitions` include basic methods for defining simple commands that are triggered by only one modality as well as a method for defining commands that are triggered by multiple modalities. The basic methods for defining commands triggered by only one modality, are all of the form `addBasic<modality>Definition()`. For each of these basic methods, the trigger is defined by arguments passed directly to the method. In addition to the trigger arguments, the only remaining argument is the desired name for the new command to be defined. The basic method then defines a new `CommandDefinition` object from the command name and trigger arguments passed into the method, automatically registers the newly defined `CommandDefinition` object with the `CommandDefinitionRegistry`, and, finally, returns a reference to the newly defined `CommandDefinition` to the caller. The more complex

`addDefinition()` method provided by the `CommandDefinitionRegistry`, requires the programmer to pass the desired name for the new command and an array of already defined `ModeTriggers`.

The remaining `generateCommandEvents()` method provided by the `CommandDefinitionRegistry` is used internally by the `MultimodalIntegrationAgent`. This method takes a `ModalContent` object resulting from any modality's input event and returns an array of `CommandEvent` objects. The returned array contains a `CommandEvent` object for each registered `CommandDefinition` that is triggered by the input event represented in the provided `ModalContent` object.

### 3.4 The *resolver* Package

The *resolver* package augments EMMET's multimodal input mode support by providing a tag resolving architecture that addresses diectic, anaphoric, and object references occurring in speech utterances. In this context, diectic references are words that specify a spacial or temporal location from the perspective of the user, anaphoric references are words, such as a pronouns, that refer back to another unit, and object references are references to objects visible to the user. This tag resolving architecture can be extended to support any number of additional speech reference types, defined by the programmer.

The tag resolving architecture provided by EMMET overloads the JSGF tag syntax, allowing programmers to request that certain information be obtained and recorded while speech utterances, derivable from the defined JSGF speech rule, are being recognized.

The following example should help to clarify this use of speech tags. Suppose a programmer wants to define a command to add ball objects to a three-dimensional

world at specific locations. He wants the user to invoke this command using speech. He would first need to define a speech trigger for this command. The JSGF speech rule string for this trigger would be "Add a ball <diectic\_ref>. The <diectic\_ref> rule is predefined by EMMET to recognize the words "here" and "there". However when this speech rule is recognized and the command is triggered, he will then need to know where the mouse cursor was when the user uttered the diectic word. To ensure that this mouse cursor location is available to him, the programmer can add a location tag to his JSGF speech rule, as in "Add a ball <diectic\_ref> {location}". Adding this tag indicates to the MultimodalIntegration agent that it should use the DiecticResolver to obtain the the mouse cursor's screen location Point at the time of the <diectic\_location> utterance and store the resolved Point value with the {location} tag. The programmer can now obtain this location from the CommandEvent object, generated by the recognition of the speech rule, via the CommandEvent.getTagContents() method.

### ***Resolvers***

In the previous example a Resolver was utilized to obtain the mouse cursor location during the time a speech utterance was made. The current set of implemented Resolver subclasses, available in the *resolver* package, include the ColorResolver, DiecticResolver, WorldObjectResolver, and the WorldObjectTypeResolver. The ColorResolver resolves a ParsedRuleNode tree, resulting from a recognized utterance of the predefined <color> grammar rule, into a corresponding ColorRGBA object which is usable by jME for coloring rendered objects. As described in the preceding example, the DiecticResolver resolves a ParsedRuleNode tree, resulting from a recognized utterance of the predefined <diectic\_ref> grammar rule, into the Point object reflecting the

location of the mouse at the time of the utterance. The `WorldObjectResolver` resolves a `ParsedRuleNode` tree, resulting from a recognized utterance describing an object visible to the user, into a `WorldObject` reference to the described object's model. Finally, the `WorldObjectTypeResolver` resolves a `ParsedRuleNode` tree, resulting from a recognized utterance of an object type, into the corresponding `WorldObjectType`.

The abstract `Resolver` class, shown in Figure 3.41, is the foundation from which all resolver implementations must be extended. Each `Resolver` subclasses must implement the abstract methods, `resolve()` and `canResolve()`. The `canResolve()` method implementation should, as efficiently as possible, return a boolean value indicating whether the `Resolver`'s `resolve()` method is capable of resolving the object argument passed into `canResolve()`. Implementing an additional `canResolve()` method, instead of simply implementing the `resolve()` method to return null when it cannot resolve an object, was necessary because some implementations of `resolve()` might want to return null as the resolved value for an input object. The implementation of `resolve()` takes an input object and returns the resolved value for that object. The abstract `resolve()` method definition has maximal genericity as both its argument and its return type are Java Objects. This allows the creation of `Resolver` subclassed to resolve (or convert) any object type into another object type. A concrete `Resolver` example is the `DiecticResolver` whose `resolver()` method resolves a `ParseRuleNode` tree representations of a diectic utterance into a screen location `Point` reflecting the mouse cursor location at the time of the utterance. Implementations of a the `resolver()` method are also required to call the protected `usedModalContent()` method for each `ModalContent` object referenced in performing the resolution.

The abstract `Resolver` class provides a number of helper methods to facilitate the implementation of `canResolve()` and `resolve()` in its subclasses. A subset of

 Resolver	
	resolverListenerList: List
	getName(): String
	addResolverListener(ResolverListener)
	removeResolverListener(ResolverListener)
	resolve(Object): Object
	canResolve(Object): boolean
	usedModalContent(ModalContent)
	getInputManager(): MultimodalInputMananger
	getInputsDuringIntervalIterator(ModalityType, TimeInterval): Iterator
	getInputsDuringIntervalIterator(ModalityType, long, long): Iterator
	getInputsDuringInterval(ModalityType, TimeInterval): ModalContentQueue
	getInputsDuringInterval(ModalityType, long, long): ModalContentQueue
	resolveMouseLocation(TimeInterval): Point

Figure 3.41: The Resolver Class

these helper methods provided access to all of the ModalContent objects that resulted from input events occurring across all modalities during a particular TimeInterval. These methods return input events in either a ModalContentQueue or an Iterator over the contents of a ModelContentQueue. Another helper method is `resolveMouseLocation()` which returns a Point reflecting the mouse cursor location at a particular time or the average mouse cursor location during a particular time interval. Finally, the `getInputManager()` helper method allows quick access to the singleton MultimodalInputManager.

### ***The ResolverRegistry***

The single ResolverRegistry stores all of the defined Resolvers for access by the programmer and the MultimodalIntegrationAgent. This registry initially contains all of the predefined Resolvers already mentioned. New Resolver subclasses should be added to the ResolverRegistry via the registry's `registerResolver` method. Access to

the Resolvers maintained in the ResolverRegistry is provided by the `getResolver()` method, which takes a resolver name String and returns the corresponding Resolver object, and by the `getResolverIterator()` method which returns an iterator over all registered Resolver objects.

### 3.5 The *model* Package

The *model* package helps to tie EMMET's multimodal input support to the three-dimension environments implemented using jME. This package defines a simple abstract layer which allows the rendered environment to share information with the Resolvers and CommandDefinition listeners.

The WorldObject class along with its mobile and static subclasses is shown in Figure 3.42. These classes provide generic wrappers for the concrete spatial used to represent objects in a jME rendered environment. The subclassing of WorldObjects as mobile and static helps to clarify what role the wrapped Spatial plays in the jME environment. A WorldObject wrapper should be created for any jME spatial that needs to be referenced by an EMMET Resolver or CommandDefinition listener. To create such wrappers, the programmer creates a new instance of WorldObject by passing the jME Spatial to the WorldObject class constructor. Upon creation, new WorldObject instances must be added to the singleton WorldObjectRegistry.

The singleton WorldObjectRegistry, shown in Figure 3.43, maintains and provides access to all of the register WorldObject instances. This registry is used extensively by the WorldObjectResolver for resolving object references made in speech utterances. The WorldObjectRegistry also provides a number of query methods. These query methods include `getLastResolveWorldObject()`, which returns the last WorldObject return by any Resolver's `resolve()` method;

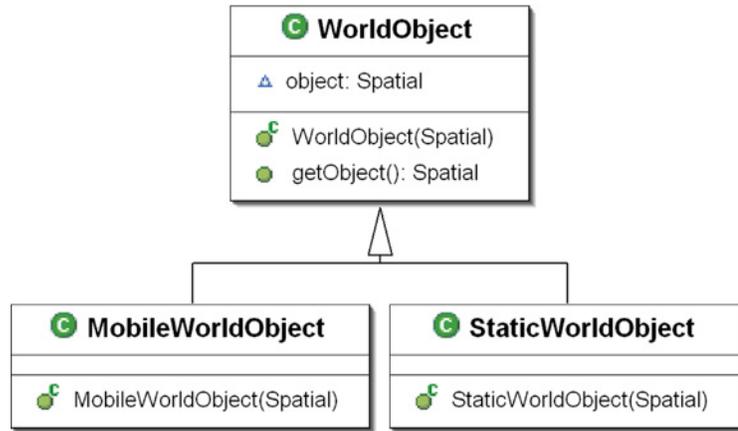


Figure 3.42: WorldObject Class with Mobile and Static Subclasses

`getRegisteredWorldObjectAtLocation()`, which returns the `WorldObject` located at a provided screen location `Point`; `getRegisteredWorldObjectByName()`, which returns the `WorldObject` with a given name; and `isRegisteredWorldObject()`, which returns whether a given `Spatial` is registered with the registry. As the `WorldObjectRegistry` extends the `AbstractModelRegistry`, note that `RegistryListeners` can be added to the `WorldObjectRegistry` to receive notification when the registry changes. `RegistryListener` interfaces defines only one callback method, `onRegistryChanged()`.

Also defined in the *model* package is the singleton `WorldObjectTypeRegistry`. This registry is use extensively by the `WorldObjectTypeResolver` for resolver object type references made in speech utterances. Each `WorldObjectType` establishes a relationship between a word and some category of object that can be rendered in a jME environment. For example a `WorldObjectType` for cubes would attach the word "cube" to a jME `Spatial` subclass defining cube objects. New `WorldObjectTypes` are created and registered via the `WorldObjectTypeRegistry`'s `registerWorldObjectType()` method.

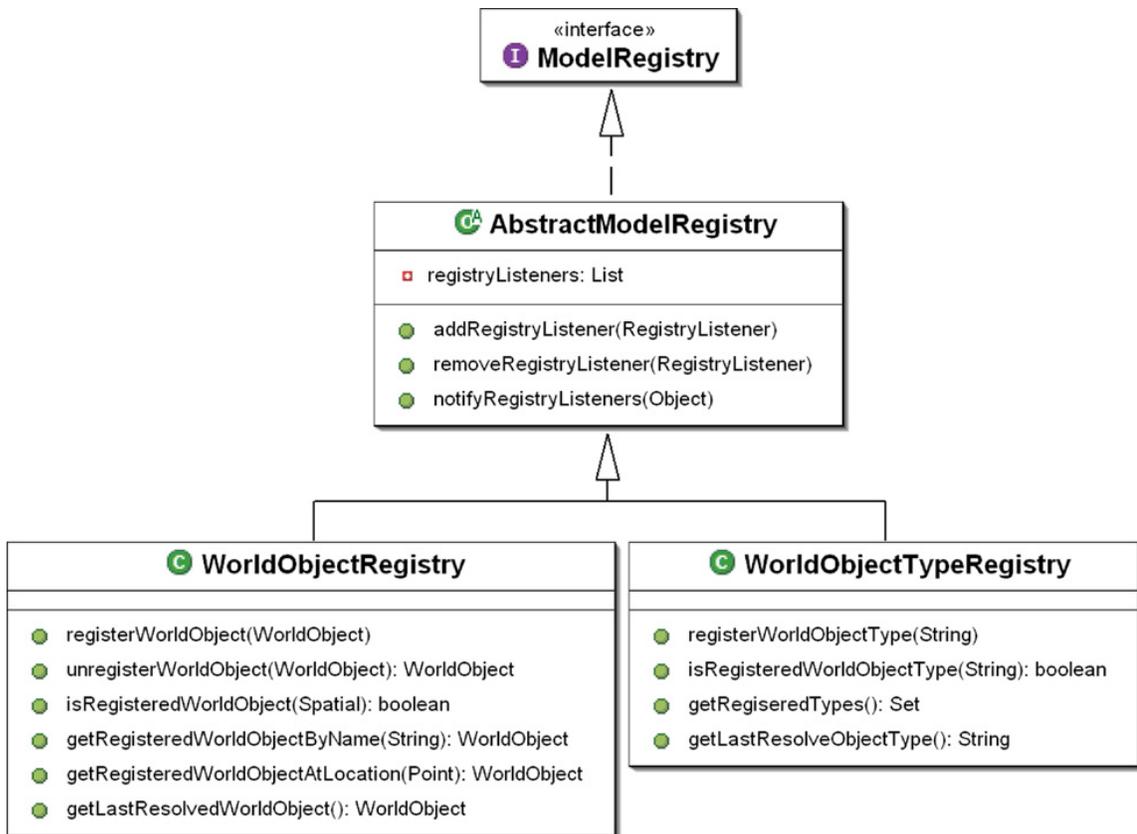


Figure 3.43: WorldObject and WorldObjectType Registries

## 3.6 The *statistics* Package

The *statistics* packages contains the classes that implement EMMET's multimodal usage statics collection and remote reporting capability. The statistics collected by EMMET include detailed information about all of the input events recognized during a user's interaction with an EMMET application. This detailed information includes the modality and associated modal content for each input event. The statistics collected also record which commands were triggered by input events, the events that triggered the commands, and the success or failure state of commands triggered. General input usage statistics recorded by EMMET's statistics collection support include the number and proportion of input events for each modality, the number of multimodal input events, and the overall percentage of successful and unsuccessful command triggers. Finally, the statistics collected include the amount and percentage of time spent using each modality.

### 3.6.1 The MultimodalStatisticsCollector

The singleton MultimodalStatisticsCollector class, shown in Figure 3.44, collects statistics on the commands called and input event triggered during a users interaction with a running EMMET application. To receive notification of all such events, the MultimodalStatisticsCollector extends the MultimodalInputAdapter class and registers itself as a listener with the MultimodalInputManager.

Upon notification of each modal input event, the statistics collector, updates the corresponding modality's input usage count and usage time. The usage count is maintained in the modalInputCount Map field, which maps a ModalityType to a total Integer value, and the usage time is maintained in the modalInputTime Map field, which maps a ModalityType to a total Long value. Upon notification of each

command triggered, via the `onCommandTriggered()` callback, a `CommandCallRecord` is instantiated. This `CommandCallRecord` is then enqueued onto a `CommandCallRecord Queue`. Each `CommandCallRecord` queue is maintained in the `MultimodalStatisticsCollector` `commandCallSetMap`, which maps `CommandDefinition` names to `CommandCallRecord` queues.

Finally, the `MultimodalStatisticsCollector` provides a `getCommandCallStatistics()` method for obtaining a report of all the statistics collected. A sample and explanation of such a statistics report is provided in chapter 4, section 3 on page 147.

The `CommandCallRecord`, shown in Figure 3.45, records the attributes of the input events that triggered a command, and the state returned from the command listener notified. Each `CommandCallRecord` includes the following:

- the name of the `CommandDefinition` called, in the `commandName` field;
- the `ModalContent` of the triggering input event, in the `triggerContent` field;
- the `ModalityType` of the triggering input event, in the `triggerModality` field;
- a `List` containing the `ModalContents` corresponding to any contributing input events, in the `modalContents` field; and
- the `CommandExitState` returned by the `CommandListener` callback that was notified of the command trigger, in the `exitState` field.

These values are all automatically set by the `CommandCallRecord`'s constructor from information obtained from `CommandEvent` and `CommandExitState` parameters.

The `CommandStatistics` class, shown in Figure 3.46, is a helper class used by the `MultimodalStatisticsCollector` `getCommandCallStatistics()` method. Each Com-

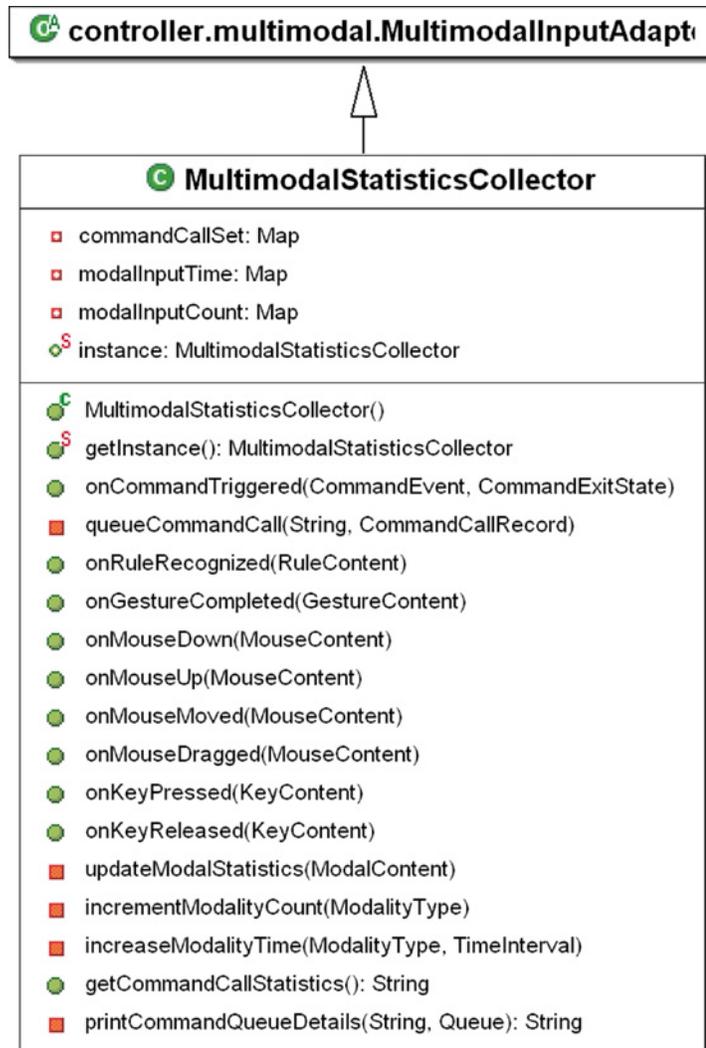


Figure 3.44: MultimodalStatisticsCollector

mandStatistics object is instantiated with a CommandCallRecord queue. The CommandStatistics class constructor uses the records in this queue to generate printable reports of successful, ignored, and failed calls to the associated command’s listeners in addition to the input event information corresponding to these calls. Thus, the `getCommandCallStatistics()` method uses a CommandStatistics instance for each CommandCallRecord queue in the MultimodalStatisticsCollector’s command-

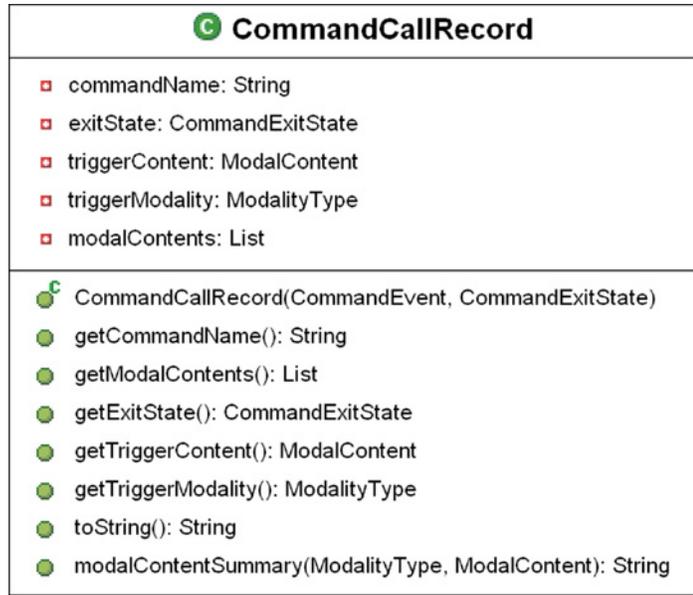


Figure 3.45: CommandCallRecord

CallSetMap.rate printable reports of successful, ignored, and failed calls to the associated command’s listeners in addition to the input event information corresponding to these calls. Thus, the `getCommandCallStatistics()` method uses a `CommandStatistics` instance for each `CommandCallRecord` queue in the `MultimodalStatisticsCollector`’s `commandCallSetMap`.

### 3.7 The *apps* Package

The *apps* package contains the `MultimodalApp` class which is the base class for creating jME applications that utilize EMMET for multimodal input support. The `MultimodalApp` class extends jME’s `FixedFramerateGame` class which creates and initializes a basic fixed frame rate jME three-dimensional rendering environment. However, `MultimodalApp` also provides a globally accessible reference the singleton `MultimodalInputManager` which it both creates and initializes. In addition, Multi-



Figure 3.46: CommandStatistics Class

modalApp creates and initialize both the WorldObjectRegistry and the WorldObject-  
TypeRegistry. Finally, MultimodalApp calls the MultimodalInputManager’s update  
method every frame to drive EMMETs multimodal input recognition processing.

# Chapter 4

## EMMET Proof of Concept Tests, Function Verification Tests, and Demonstration Applications

### 1 Pre-EMMET Proof of Concept Tests

Early research into the development of EMMET involved the creation of several proof-of-concept test cases. These test cases were created to explore possible solutions to a number of unresolved issues integral to the implementation of EMMET. The resolution of these issues required answers to the following questions:

- What technologies support the required speech and gesture recognition capabilities?
- For those technologies that meet those required capabilities, can they be used simultaneously, and can they be easily packaged with an application to be

launched remotely?

- Which 3D rendering environment should be used?
- Which 3D rendering environments can be easily packaged with an application and launched remotely?
- How should applications be made launchable from the Web?

## 1.1 Early Speech Recognition Tests

### 1.1.1 Hello World Speech Applet

One early proof-of-concept test explored Java speech recognition technology. This test required the creation of a simple *Hello Speech World* Java applet that recognized the utterances of three salutations: “Hello World,” “Hello Computer,” and “Good Morning.” Upon the utterance of any one of these phrases, the applet would display the text of the utterance, thereby confirming its speech recognition. This applet utilized the IBM JavaSpeech SDK, which includes both a speech recognition engine and an implementation of the Java Speech API (JSAPI). The *Hello Speech World* applet implementation experience provided an introduction to the Java Speech API and placed this API atop the list of possible speech recognition solutions. This test demonstrated that that speech recognition could be included in interfaces to Web accessible Java applets.

### 1.1.2 PollyWorld Layout Tester

A subsequent proof of concept test that addressed speech recognition technologies was the *PollyWorld Layout Tester*. The goal for this test was to build a speech recognition

interface that allowed users to direct the behavior and alter the appearance of simple anthropomorphic, wedge-shaped characters known as “Pollys.” These Pollys and the world in which they existed were rendered using Ken Perlin’s Java 3D renderer being run in a Web accessible Java Applet. Figure 4.1 is a screen capture of the *PollyWorld Layout Tester* applet.

In this applet, users could utter the following phrases to produce the associated results:

“**Name Polly**” Names the currently selected Polly. The currently selected Polly is the one most recently added, unless the user selected a different Polly by clicking on it with the mouse.

“**Color <Polly name> <color>**” Colors the addressed Polly the requested color. The available colors are red, blue, green, purple, yellow, or orange.

“**Tell <Polly name> to <action>**” Tells the addressed Polly to perform the requested action. The available actions are stop, idle, scamper, swagger, broad-jump, prowl, lumber, dejected, no, yes, dance, hotfeet, run, sprint, hop

The *PollyWorld Layout Tester* confirmed that the JSAPI was indeed a viable solution for the implementation of speech recognition support in EMMET. The tester also demonstrated an ability to support multimodal speech recognition with mouse input, and speech recognition with a dynamic grammar. However, a discouraging discovery made from both the *Hello Speech World* and *PollyWorld Layout Tester* tests was that Java applications and applets utilizing the JSAPI were dependent on the presence of a local speech recognition engine. Also, as the JSAPI implementation used in the development of these test cases was IBM’s JavaSpeech SDK, there existed

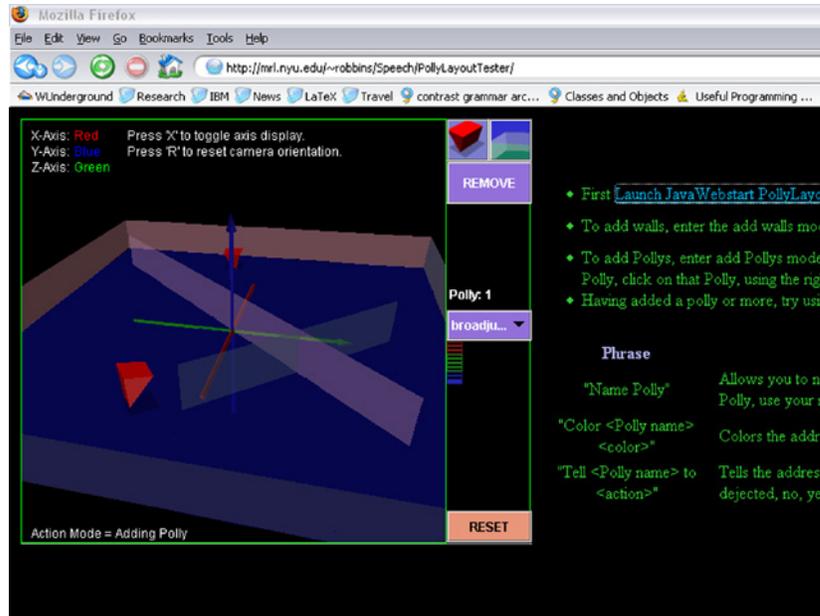


Figure 4.1: PollyWorld Layout Tester Java Applet with Speech Recognition

a more specific requirement that the local speech recognition engine be from the commercial IBM ViaVoice product line. Thus, while utilizing this technology would prove useful in developing EMMET, it would not prove practical. Because ViaVoice is a commercial product, utilizing it in EMMET would rely on the users' ownership of ViaVoice and would, therefore, diminish the availability of EMMET developed applications.

However, further research revealed a number of JSAPI implementations and eventually led to the discovery of CloudGarden's JSAPI implementation, the TalkingJava SDK. As described earlier in the discussion of EMMET's Foundational Technologies, section 2.1, the TalkingJava SDK supports a variety of speech recognition engines. Most importantly, it supports Microsoft's speech recognition engines, which are now widely accessible as they are packaged with the prevailing Microsoft operating systems, Windows 2000 and WindowsXP.

Furthermore, these early proof-of-concept tests provided assurance that speech recognition could be integrated into interfaces to Web accessible 3D rendered environments and used in conjunction with conventional keyboard and mouse input.

## 1.2 Speech and Gesture Based Geometry Placer

The previously described speech recognition tests helped to answer some of the unresolved issues integral to the implementation of EMMET. However, the *Speech and Gesture Based Geometry Placer* was a proof-of-concept test intent on categorically resolving all of the remaining issues such that the implementation of EMMET could commence. This geometry placer test was the culmination of a long series of tests that were each implemented to address some subset of the remaining unresolved issues.

The comprehensive goal of the geometry placer was that it exhibit all of the capabilities desired for applications that would eventually be developed using EMMET. Thus the geometry placer demo was to be a Web launchable application with a multimodal speech and gesture based interface to a 3D rendered environment. Such a solution would have to resolve what technologies are required to implement an application that supports the simultaneous use of speech and gesture recognition, and can easily be packaged and launched remotely via the Web. The resulting implementation of the *Speech and Gesture Based Geometry Placer* did indeed provided these answers.

The geometry placer first established the solution for symbol gesture recognition by successfully utilizing the HHReco Graphics Symbol Recognition Toolkit, described in the EMMET Foundational Technologies section. The geometry placer used this toolkit to allow users to add various objects to a finite planer area in a 3D rendered environment. Users could make a circular gesture to create a new ball object, a square gesture to create a new cube object, a triangle gesture to create a new cone object,

and a Christmas tree like gesture to create a new tree object. The location of these objects was determined by calculating the center point of the gesture drawn on the screen and projecting that point onto the planar floor upon which the objects were added. Note that if the ray projected from the calculated center point intersected an existing object prior to reaching the floor, then the new object was rooted at the intersection point instead. An additional non-symbolic gesture recognition was added in which the user could draw a slash mark across any of the existing objects to request that they be erased. Determining exactly which object the user intended to erase was similarly resolved by projecting the center of the slash mark until it intersected with an existing object. The scale of the added object was also proportional to the size of the gesture drawn. The successful implementation of these gesture recognition interface capabilities for the geometry placer confirmed that the HHReco toolkit was sufficient for implementing the gesture recognition support in EMMET.

Similar to the early speech recognition tests, the JSAPI was used to code the speech recognition for the geometry placer. However, the geometry placer test used the Cloudgarden's TalkingJava SDK implementation of the JSAPI and used the Microsoft speech recognition engine packaged with current Microsoft operating systems. The speech recognition capabilities for the geometry placer allowed the user to utter phrases such as "Add a <object\_type> here" to request that a cube, ball, or cone object be added at the location indicated by the current mouse cursor. The user could also utter the phrase "Erase this" to remove the object under the current mouse cursor location. A rudimentary dialog history was used to allow user to say "undo that" or "erase it" and resolve the anaphor to the last thing done or the last thing added. Finally, users could speak the phrase, "Color this <color>" to request that the object located under the current mouse point cursor be colored the requested

color. This successful implementation of this speech recognition confirmed that the use of Cloudgarden's JSAPI couple with the built-in Microsoft speech recognition engine was a sufficient solution for the implementation of speech recognition support in EMMET.

The 3D rendering environment used for the geometry placer was Ken Perlin's 3D renderer running in OpenGL hardware accelerated mode. The successful simultaneous use of this rendering environment, HHReco gesture recognition, and Cloudgarden's speech recognition confirmed that these three technologies could be used together to implement EMMET's full multimodal interface support.

Finally, a WebStart JNLP script was configured to launch the *Speech and Gesture Based Geometry Placer* along with all resources required for the rendering environment, and speech and gesture recognition support. The geometry placer's successful launch from the Web using the JNLP file confirmed that all of the confirmed technologies could be used to implement EMMET applications that could be launched via the Web. Figure 4.2 shows a screen capture of an interaction session with the geometry placer as it appeared upon being launched via the Web.

## **2 EMMET Development Function Verification Tests**

During the development of EMMET a number of milestones were reached that marked the completion of logical units of functionality. Each of these units encompassed the implementation of a particular feature supported by EMMET:

1. use of each modality running in unimodal input mode,

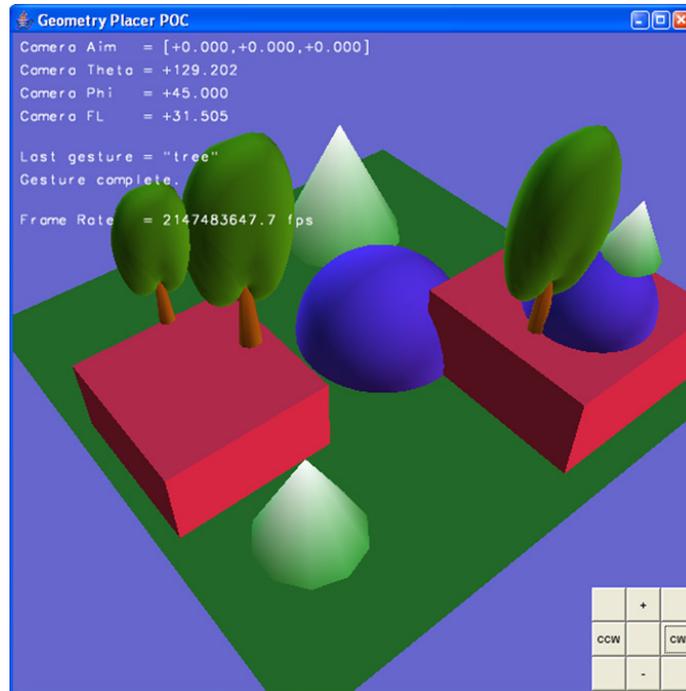


Figure 4.2: Speech and Gesture Based Geometry Placer Proof-Of-Concept

2. use of multiple modalities running in unimodal input mode,
3. use of commands and triggers, and hence use of multimodal input mode,
4. use of multimodal usage statistics collection,
5. remote use via the Web, and
6. use of remote statistics collection and reporting via the Web.

As implementation of the early features listed in item one reached completion, a concise test case was developed to ensure that each feature functioned correctly. Thus function verification test cases were developed for testing EMMET's unimodal input mode support for mouse and keyboard handling, gesture recognition, and speech recognition. Next, a function verification test was developed for testing feature item

two, EMMET’s support for the combined use of these modalities in unimodal input mode. The remaining features listed in items three through six were function tested in progressive steps that were made in the development of a final EMMET version of the Pre-EMMET *Speech and Gesture Based Geometry Placer* application.

## 2.1 EMMET Unimodal Input Mode Test Applications

The function test cases used to verify the implementation of the early features supported by EMMET are included in the *tests* package. These test cases include TestKeyboardInput, TestMouseInput, TestGestureInput, and TestSpeechInput. These tests were critical in ensuring the completeness and correctness of EMMETs support for each modality before moving on to the next. Each test also verified that EMMET’s use of the underlying technology was successful.

### 2.1.1 Keyboard and Mouse

The first two test cases, TestKeyboardInput and TestMouseInput, validated the implementation of keyboard and mouse support in EMMET. These test cases are grouped together because EMMET’s support for both of these modalities utilizes the conventional input handling interface provided by jME. However, the purpose of providing replacement EMMET implementations of these modalities, as has already been discussed, is to align with EMMETs overarching goal of providing a consistent input handling interface across all modalities.

Figures 4.3 and 4.4 contain code excerpts from TestKeyboardInput and TestMouseInput. Each excerpt shows the instantiation of the respective modality’s singleton input manager and the registration of one or more simple input handlers with that input manager. The code for each callback implemented in these simple handlers

merely prints to standard output the contents of the ModalInput parameter passed to them. For all of unimodal input mode tests, the 3D rendered environment is a placeholder terrain generated from a height map.

```
KeyboardInputManager keyboardManager =
    KeyboardInputManager.getKeyboardManager(input);

keyboardManager.addKeyHandler(new KeyHandler()
{
    public void onKeyPressed(KeyContent c)
    {
        System.out.println("Key Pressed: " + c);
    }

    public void onKeyReleased(KeyContent c)
    {
        System.out.println("Key Released: " + c);
    }
});
```

Figure 4.3: TestKeyboardInput Excerpt

Note the similarity between EMMET's interfaces for conventional modality input handling and other commonly used interfaces for such input handling like Swing and AWT. This similarity is intentional and evolved from the belief that EMMET's conventional input modality interfaces could be both familiar to users of existing input handling interfaces, and, yet, consistent with EMMET's additional speech and gesture input handling interfaces.

```

MouseListener mouseManager =
    MouseInputMananger.getMouseManager(input);

mouseManager.addMouseListener(new MouseHandler() {

    public void onMouseDown(MouseContent c)
    {
        System.out.println("MouseDown: " + c);
    }

    public void onMouseUp(MouseContent c)
    {
        System.out.println("MouseUp: " + c);
    }
});

mouseManager.addMouseMotionHandler(new MouseMotionAdapter() {

    public void onMouseDragged(MouseContent c)
    {
        System.out.println("MouseDragged: " + c);
    }

    public void onMouseMoved(MouseContent c)
    {
        System.out.println("MouseDragged: " + c);
    }
});
}

```

Figure 4.4: TestMouseListener Excerpt

### 2.1.2 Gesture

The TestGestureInput application parallels the structures of TestKeyboardInput and TestMouseInput to verify EMMET's gesture recognition input handling interface. The relevant TestGestureInput excerpt is shown in Figure 4.5. Similar to the keyboard and mouse test cases, a GestureHandler, in which each callback implementation simply prints the input parameters, is registered with the singleton GestureInputManager. Prior to registering the Gesture Handler, however, the TestGestureInput application also calls upon the gesture manager to load an HHReco gesture training file. This file was created using the HHReco toolkit and contained training samples for shapes such as a star, cloud, tree, and squiggle.

```
GestureInputManager gestureManager =
    GestureInputManager.getGestureManager();

gestureManager.loadGestureTrainingFile(
    getClass().getResource("data/gesture/simple_gestures.sml"));

gestureManager.addGestureHandler(new GestureHandler()
{
    public void onGestureBegun(int x, int y)
    {
        System.out.println("Gesture Begin: (" + x + "," + y + ")");
    }

    public void onPointAdded(int x, int y)
    {
        System.out.println("Gesture Point Added: (" + x + "," + y + ")");
    }

    public void onGestureCompleted(GestureContent c)
    {
        System.out.println("Gesture Complete: " + c);
    }
});
```

Figure 4.5: TestGestureInput Excerpt

TestGestureInput verifies EMMET’s unimodal gesture recognition support, first by ensuring that user drawn gestures are accurately recognized by the GestureInput manager, second by ensuring that the results of the recognition are properly converted into a GestureContent object, and third by ensuring that all registered GestureHandler implementations are properly notified and passed the generated Gesture object. The screen grab shown in Figure 4.6, is of a running TestGestureInput application in which the user is drawing a tree gesture object.

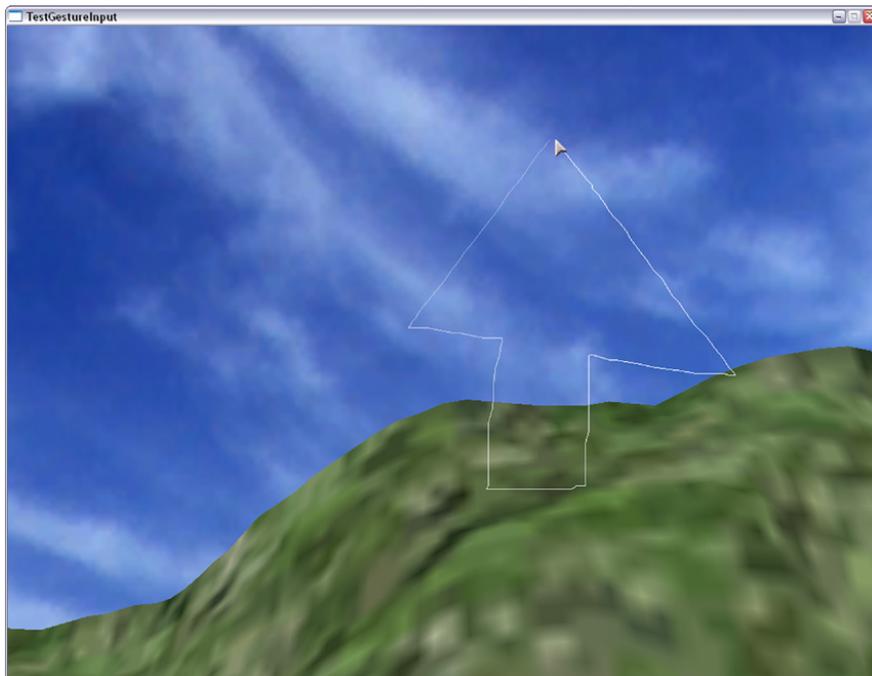


Figure 4.6: TestGestureInput Application with User Drawing a Tree Gesture

### 2.1.3 Speech

The final installment of the individual modal input manager test cases is TestSpeechInput. For this test case, a number of RuleHandlers were bound to dynamically defined speech rules allowing the user to exit the test, regenerate the terrain, and

look around the world by turning to the left or right. Figure 4.7 is an excerpt from `TestSpeechInput` showing the instantiation of the speech input manager following by the code for defining and registering each speech rule. `TestSpeechInput` verifies the EMMET implementation of unimodal speech recognition, by ensuring that new speech rules could be dynamically defined and added to the `SpeechInputManagers` grammar, and that `RuleHandlers` could be defined and registered to receive notification when certain speech rules are recognized.

#### **2.1.4 Combined Speech and Gesture in Unimodal Input Mode**

The prior testcases verified the implementation of each EMMET modal input manager when used individually in unimodal input mode. The next step was to verify that these input managers could all be instantiated and coexist in one application that simultaneously allowed any modality to be used in unimodal input mode.

To provide the aforementioned verification, a test case aggregate of the individual modal input testcases was developed called `TestAllModalInputs`. In `TestAllModalInputs`, the user can provide keyboard, mouse, gesture, or speech input. Also, the instantiation of each input manager and both the creation and registration of each manager's corresponding input handlers was derived from the individual modal input test cases. Thus in `TestAllModalInputs`, the response to each user input is the same as the response implemented for the same input in its associated individual modal input test case.

For example, a user of `TestAllModalInputs` can draw a symbol gesture and witness the response implemented in `TestGestureInput`, which was to see the resulting recognized symbol name printed to standard output. The user can also utter the phrase, "Turn left", to witness the `TestSpeechInput` response of turning the viewing

```

SpeechInputManager speechManager = SpeechInputManager.getSpeechManager();

speechManager.addRule("exitTest", "Exit Test", new RuleHandler()
{
    public void onRuleRecognized(RuleContent c)
    {
        System.out.println(c); finish();
    }
});

speechManager.addRule(
    "NewTerrain", "[Generate] New Terrain", new RuleHandler()
{
    public void onRuleRecognized(RuleContent c)
    {
        System.out.println(c); regenerateTerrain();
    }
});

speechManager.addRule("turnLeft", "Turn Left", new RuleHandler()
{
    public void onRuleRecognized(RuleContent c)
    {
        System.out.println(c); actionInProgress = LEFT;
    }
});

speechManager.addRule("turnRight", "Turn Right", new RuleHandler()
{
    public void onRuleRecognized(RuleContent c)
    {
        System.out.println(c); actionInProgress = RIGHT;
    }
});

speechManager.addRule("turnStop", "Stop turning", new RuleHandler()
{
    public void onRuleRecognized(RuleContent c)
    {
        System.out.println(c); actionInProgress = NONE;
    }
});

```

Figure 4.7: TestSpeechInput Excerpt

camera to the left.

TestAllModalInputs verified the combined use of EMMET's modal input managers running in unimodal input mode. Such verification was essential before an attempt could be made to implement EMMET's true multimodal support which would allow cross-modal interpretation of input events and input event history lookup.

## 2.2 EMMET Multimodal Input Mode Test Application

The final test case developed, prior to work on a full EMMET demonstration application, was `TestMultimodalInput`. This test case moved into the use of the `MultimodalInputManager` as the central access point for input event handling. Thus the instantiation of all the single input managers was left to the `MultimodalInputManager`. Also, calls to commonly used methods in any individual input managers, could be made via the convenience methods provided by the `MultimodalInputManger`. In addition, a reference to the `MultimodalInputManager` was readily available, as it is already instantiated and initialized by the `MultimodalApp` base class from which `TestMultimodalInput` was derived. This reference to the `MultimodalInputManager` is the `emmetInputHandler` variable and is available to any multimodal application that extends the `MultimodalApp` base class.

The `TestMultimodalInput` test case also utilizes the command and triggers methodology introduced for multimodal input mode event handling. Thus, simple speech rules are defined and activated by using the `CommandDefinitionRegistry`'s `addBasicSpeechDefinition()` method. Figure 4.8 shows the definition of a new speech command for responding to the user's utterance of the phrase "Test Speech". Note that `addBasicSpeechDefinition()` is a convenience method that performs a number of tasks in the background. For example, the following tasks are performed when `addBasicSpeechDefinition()` is called in the code excerpt:

1. create new `Command` named "TestSpeech",
2. define a new grammar rule called `<testSpeech>` that recognizes the utterance "Test Speech",
3. create a new `SpeechRuleTrigger` that is triggered by the recognition of the

<testSpeech> speech rule, and

4. register the new “TestSpeech” Command with the CommandDefinitionRegistry.

The CommandDefinitionRegistry’s convenience methods for defining new commands return the new CommandDefinition object defined. Therefore, in the code excerpt, the new CommandDefinition object returned by `addBasicSpeechDefinition()` is used to call `addCommandListener()`, which is passed a new CommandAdapter argument that responds when the “Test Speech” command is triggered.

```
CommandDefinitionRegistry.  
addBasicSpeechDefinition("TestSpeech", "testSpeech", "Test Speech").  
addCommandListener(  
    new CommandAdapter()  
    {  
        public CommandExitState onSpeechTrigger(RuleContent content,  
                                                CommandEvent event)  
        {  
            System.out.println{"\"Test Speech\" Recognized"};  
            return CommandExitState.SUCCEEDED;  
        }  
    }  
);
```

Figure 4.8: Basic Speech Definition Use of Command and Trigger

The code excerpt in Figure 4.9 shows the implementation of a “Look left” CommandDefinition. In the code excerpt a new CommandDefinition is being added to the CommandDefinitionRegistry via a call to `addDefinition()`. This command and trigger implementation mimics the <turnLeft> SpeechRule creation and RuleHandler registration performed in TestSpeechInput.

Similar to `addBasicSpeechDefinition()`, the `addDefinition()` method performs many tasks in the background for the programmer. However, `addDefinition()`

```

CommandDefinitionRegistry.addDefinition(
    "LookLeft",
    new ModeTrigger[] {
        new SpeechRuleTrigger("lookLeft", "Look Left"),
        new KeyboardTrigger(Keyboard.KEY_LEFT, KeyInputType.KEY_DOWN)
    }).addCommandListener(
    new CommandAdapter()
    {
        public CommandExitState onCommandTrigger(ModeContent content,
                                                    CommandEvent event)
        {
            actionInProgress = TURNING_LEFT;
            return CommandExitState.SUCCEEDED;
        }
    });

```

Figure 4.9: Look Left Definition Responds to Speech or Keyboard Triggers

requires the programmer to manually create the ModeTriggers for the Command and to pass those triggers in an array argument. This requirement is necessary to allow the addition of an indefinite number of command triggers from multiple modalities to the command definition being defined. Thus, in the TestMultimodalInput code excerpt, both a SpeechRuleTrigger and a KeyboardTrigger are defined and added to the same CommandDefinition. The SpeechRuleTrigger defines a new grammar rule called <lookLeft> which is fired by an utterance of the phrase “Look Left”, while the KeyboardTrigger is defined to respond when the left arrow key is down.

Upon creating the “LookLeft” CommandDefinition and adding both speech and keyboard ModeTriggers, a CommandListener is added to the CommandDefinition. The command listener implements the generic onCommandTrigger() method, that responds to the firing of either ModeTrigger, to set the enclosing application’s actionInProgress state variable to TURNING\_LEFT. Because there are no tag or modal content references required by the CommandListener, the call to onCommandTrigger()

is assumed to have been successful and, hence, `CommandExitState.SUCCEEDED` is returned.

The `TestMultimodalInput` code excerpt in Figure 4.10 illustrates how one would utilize multimodal input mode to color objects in a rendered world. The “ColorSomething” Command Definition being created in the excerpt defines a command that required cross-modal interpretation of both mouse and speech input. To achieve this cross-modal interpretation, tags are included in the speech rule definition and are later referenced to resolve the object being diectically indicated by the mouse cursor.

```
CommandDefinitionRegistry.addBasicSpeechDefinition(  
    "ColorSomething",  
    "colorSomething",  
    "color <object_ref> {world_object} <color> {color}").addCommandListener(  
        new CommandAdapter()  
        {  
            public CommandExitState onSpeechTrigger(RuleContent content,  
                                                    CommandEvent event)  
            {  
                System.out.println("color "  
                    + event.getTagContents("world_object").getClass().getName()  
                    + " "  
                    + (ColorRGBA)event.getTagContents("color"));  
                return CommandExitState.SUCCEEDED;  
            }  
        }  
    });
```

Figure 4.10: ColorSomething Speech Definition Responds to Multimodal Input

The “colorSomething” `SpeechRuleTrigger` is defined and added to the “ColorSomething” `CommandDefinition` as described earlier in the “TestSpeech” explanation. However, the `CommandListener` registered with the “ColorSomething” `CommandDefinition` implements `onSpeechTrigger()` to additionally take advantage of the built in tag support provided by multimodal input mode. With this tag sup-

port, EMMET’s `MultimodalIntegrationAgent` internally resolves the `{world_object}` tag into the object indicated by the user during the utterance of the `<object_ref>`. For testing purpose, `onSpeechTrigger()` simply obtains and displays the resolved `{world_object}`’s class along with the resolution of a second tag, `{color}`, which is resolved to the color specified by the `<color>` utterance.

Note that a `CommandDefinition`’s dependence upon input from multiple modalities does not entail a need for multiple triggers. Triggers only define the input events that trigger or initiate the recognition of a command. For example, the “Color-Something” `CommandDefinition` depends on both speech and mouse input. Yet the definition only has a trigger for speech because it is the utterance of the phrase that triggers the command, not any input from the mouse.

Also note that, for simple function verification testing, the `CommandListener` defined in the code excerpt always returns `CommandExitState.SUCCEEDED`. However for accurate statistics collection and multimodal input mode use, the conscientious programmer should consider returning `CommandExitState.FAILED` if either of the tag contents are null, or possibly returning `CommandExitState.IGNORED` if the `{world_object}` tag resolves to a world object that can not be colored.

The `TestMultimodalInput` function test verified EMMET’s implementation of the command-and-triggers architecture and multimodal input mode. In addition to the discussed code excerpts, the following `CommandDefinitions` were defined in `TestMultimodalInput` to verify other aspects of EMMET’s support:

- a `CommandDefinition` triggered by inputs from all modalities,
- a `CommandDefinition` using combined input from both speech and gesture,
- speech triggered `CommandDefinitions` dependent on multiple tags requiring

mouse location or world object resolution,

- at least one CommandDefinition for each CommandDefintion creation convenience method provided by the CommandDefinitionRegistry, and
- a CommandDefinition utilizing input event history support.

### 3 EMMET Geometry Placer Demonstration Application

Upon verifying EMMET's implementation of unimodal and multimodal input mode support, work commenced on the development of an application to exhibit the previously verified support as well as the support provided by EMMET for remotely launching EMMET applications and for remote statistics collection. This exhibitory application utilized EMMET to rapidly reproduce and also enhance the pre-EMMET *Speech and Gesture Based Geometry Placer* proof-of-concept test.

Figure 4.11 shows a screen capture of a user drawing a square gesture in the EMMET Geometry Placer application. In the EMMET Geometry Placer the user interacts with a 3D rendered world in a manner similar to that of the pre-EMMET version. However, the EMMET Geometry Placer uses a jME sky box to create a mountainous feel for that world and a transparent plane, floating above, to provide the floor upon which objects are created. The purpose of the sky box is to establish that the underlying jME 3D renderer, upon which EMMET is built, can produce rich and sometimes complex content. In actuality, the environment used for the EMMET Geometry Placer merely hints at the capabilities of the jME 3D render.

Figure 4.12 shows the EMMET Geometry Placer registering the types of ob-

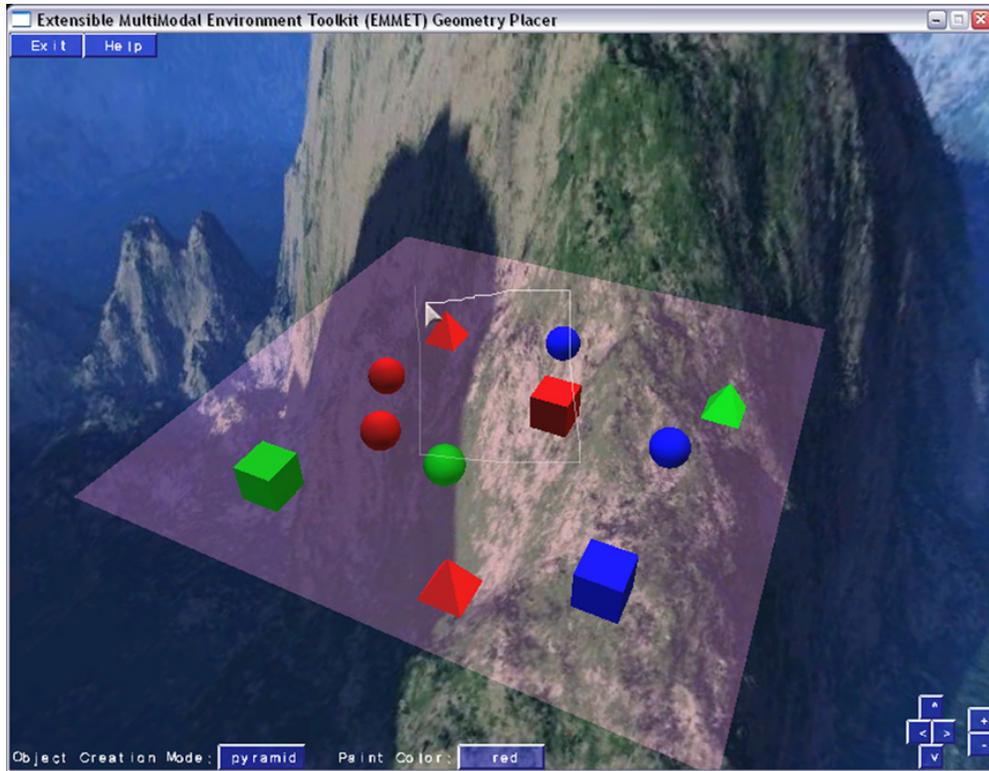


Figure 4.11: EMMET Geometry Placer

jects that can be created in the world. These object types are registered with the `WorldObjectTypeRegistry` and are used by the `WorldObjectTypeResolver` to resolve tagged occurrences of the `<object_type>` rule when used in defining speech rules. The Geometry Placer next registers the root node of the 3D modeled scene with the `WorldObjectRegistry`. This root node is used by the `WorldObjectResolver` in traversing the scene graph to resolve the tagged occurrence of the `<object_ref>` rule when used in defining speech rules.

```
// Register world object types with the world object type registry
WorldObjectTypeRegistry.getInstance().registerWorldObjectType("ball");
WorldObjectTypeRegistry.getInstance().registerWorldObjectType("pyramid");
WorldObjectTypeRegistry.getInstance().registerWorldObjectType("cube");

// Initialize the world object registry with this world's root scene node
WorldObjectRegistry.initalizeInstance(rootNode);
```

Figure 4.12: Registering World Object Types and the `WorldObjectRegistry`

The code excerpts in Figures 4.13 and 4.14 show the Geometry Placer’s use of EMMET’s full multimodal input mode capabilities. The first code excerpts shows the definition and registration of an “addNewObject” `CommandDefinition` for adding objects to the 3D rendered world. Triggers are defined for “addNewObject” that respond to input from any of EMMET’s supported modalities.

Two speech rule triggers are defined for “addNewObject.” The first speech rule, “addObject,” recognizes utterances in which the user requests that an object of a certain type be added to the 3D world at a diectically indicated location on the planar floor. The user may optionally specify what color the new object should be. Note that the object types recognized by this speech rule are the types that were registered with the `WorldObjectTypeRegistry`. Tags are also defined in the first speech rule to

```

CommandDefinitionRegistry.addDefinition(
    "addNewObject",
    new ModeTrigger[]
    {
        new SpeechRuleTrigger(
            "addObject",
            "Add [(a|an)] [<color> {color}] <object_type> {object_type}
            <diectic_ref> {location}"),

        new SpeechRuleTrigger("addAgain", "and <diectic_ref> {location}"),

        new GestureTrigger("ball", GestureType.SYMBOL_GESTURE),
        new GestureTrigger("cube", GestureType.SYMBOL_GESTURE),
        new GestureTrigger("pyramid", GestureType.SYMBOL_GESTURE),

        new KeyboardTrigger(Keyboard.KEY_B, KeyInputType.KEY_DOWN),
        new KeyboardTrigger(Keyboard.KEY_C, KeyInputType.KEY_DOWN),
        new KeyboardTrigger(Keyboard.KEY_P, KeyInputType.KEY_DOWN),

        new MouseTrigger(MouseInputType.MOUSE_DOWN,
            MouseButtonStateType.MOUSE_BUTTON_2)
    })

```

Figure 4.13: Geometry Placer “addNewObject” CommandDefinition Triggers

allow “addNewObject” CommandListeners to retrieve the color, object type, and screen location specified. The second speech rule, “addAgain,” allows the user to use phrases such as “and here” or “and there” to request the creation of another instance of the same object type specified in a prior utterance of “addObject.” A CommandListen

Three gesture rule triggers are also defined for the “addNewObject” Command-Definition. Each of these gesture triggers responds to the recognition of a symbol gesture that is associated with a certain world object type. Hence, gesture triggers are defined to recognize a circle gesture, to add a new ball; a square gesture, to add a new cube; and a triangle gesture, to add a new pyramid.

Similarly, three keyboard triggers are defined for “addNewObject”. These triggers allow the user to press the ‘B’, ‘C’, or ‘P’ key to respectively add a new ball, cube, or pyramid object.

Finally, a mouse trigger is defined for “addNewObject” to add a new object when the user presses the right (or second) mouse button.

The code excerpt in Figure 4.14 shows the `CommandListener` implemented to handle the triggering of “addNewObject” command events. Although each callback implemented in the listener ultimately calls the `addObject()` method, passing object type and mouse location parameters, a modal callback for each modality was required. This requirement stems from differences in the means used to obtain the object type and mouse location parameters. For example, the mouse location is obtained in the `onGestureTrigger()` callback on line 5 from the drawn gesture’s center point, and obtained in the `onSpeechTrigger()` callback from the `{location}` tag contents on line 27.

A number of seemingly unusual steps are taken in the implementation of `onMouseTrigger()`. The steps on lines 18–19 ensure that the current mouse cursor location is not over a graphic user interface (GUI) widget, such as the “Object Creation Mode” button. If the mouse cursor *is* over a GUI button then the triggering of the command event should be ignored. The step on line 20 that follows is used to obtain the `objectType` from the current state of the “Object Creation Mode” button. Lines 21–22 ignore the trigger when the “Object Creation Mode” state is “erase.”

The “addNewObject” `CommandListener`’s `onSpeechTrigger()` callback is implemented in lines 25–34. As previously mentioned, the mouse location corresponding to the deictic reference, made in the speech utterance, is obtained from the `{location}` tag contents. The object type specified in the utterance is dependent on which speech

```

1  new CommandListener()
2  {
3      public CommandExitState onGestureTrigger(GestureContent content, CommandEvent event)
4      {
5          Point mouseLocation = content.getGestureCenterPoint();
6          String objectType = content.getGestureSymbol();
7          return CommandExitState.booleanToSuccessOrFailure(addObject(objectType, mouseLocation));
8      }

9      public CommandExitState onKeyTrigger(KeyContent content, CommandEvent event)
10     {
11         Point mouseLocation = event.getTriggerMouseLocation();
12         String objectType = keyCodeToObjectName(content.getKeyCode());
13         return CommandExitState.booleanToSuccessOrIgnore(addObject(objectType, mouseLocation));
14     }

15     public CommandExitState onMouseTrigger(MouseContent content, CommandEvent event)
16     {
17         Point mouseLocation = content.getLocation();
18         if (uiManager.isUIManagedObjectAt(JMEUtil.awtToJME3D(mouseLocation)))
19             return CommandExitState.IGNORED;

20         String objectType = modeButton.getCurrentMode();
21         if (objectType.equalsIgnoreCase("erase"))
22             return CommandExitState.IGNORED;

23         return CommandExitState.booleanToSuccessOrIgnore(addObject(objectType, mouseLocation));
24     }

25     public CommandExitState onSpeechTrigger(RuleContent content, CommandEvent event)
26     {
27         Point mouseLocation = (Point)event.getTagContents("location");

28         String objectType = null;
29         if (content.isInstanceOfRule("addAgain"))
30             objectType = WorldObjectTypeRegistry.getInstance().getLastResolveObjectType();
31         else
32             objectType = (String) event.getTagContents("object_type");

33         return CommandExitState.booleanToSuccessOrFailure(addObject(objectType, mouseLocation));
34     }
35 }

```

Figure 4.14: Geometry Placer “addNewObject” CommandDefinition Listener

rule trigger occurred. For the “addAgain” speech rule, the object type is the last resolved object type from the `WorldObjectTypeRegistry` as shown in line 30. Otherwise, the trigger was “addObject” and the object type is obtained from the `{object_type}` tag.

The Java Network Launching Protocol (JNLP) file used to launch the EMMET

Geometry Placer via Java™ WebStart is shown in Figure 4.15. This JNLP file is included with EMMET and can be used as a template for launching future EMMET based applications. The required resources and JARs referenced by the JNLP file pertain to the HHReco Toolkit, jME, CloudGarden’s TalkingJava SDK, and the JavaMail API. The one native library resource in the JNLP file includes the DLLs required for jME and CloudGarden. Some resources can be omitted depending on which EMMET features are used: the activation, mailapi, and smtp JAR files are only required for remote statistics collection; the cgjsapi JAR is only required if speech recognition is utilized; and the hhreco JAR is only required if gesture symbol recognition is utilized.

While the Geometry Placer application is running, EMMET’s statistic collection engine is constantly accumulating data about modal input events, command triggers, and command listener calls. By default, this statistics collection is automatically engaged and requires no additional code in EMMET applications. When launched via the Web, an application can activate the remote reporting of these statistics just prior to exiting the application as shown in Figure 4.16. The SMTPResultMailer sends the collected statistics to the e-mail address specified during its instantiation.

An excerpt from the statistics report sent from an interaction with the Geometry Placer is shown in Figure 4.17. This excerpt shows the usage statistics for the aforementioned “addNewObject” command. These statistics provide a detailed account of which modalities and combinations of modalities were used to add objects and with what degree of success. Note that both of the SpeechRuleTriggers for “addNewObject” utilized tags and thus any successful triggering of the command by speech would be considered multimodal. The details pertaining to multimodal speech usage also include the resolved values for each tag. Additionally, note that the KeyboardTriggers for “addNewObject” required the mouse location in order to place the objects

```

<?Xml version="1.0" encoding="utf-8"?>
<!-- EMMET API Demo via Web Start -->
<jnlp spec="1.0+" codebase="http://cat.nyu.edu/~robbins/EMMET/" href="EMMET.jnlp">
  <information>
    <title>EMMET API GeometryPlacer Demo</title>
    <vendor>Christopher Robbins</vendor>
    <homepage href="http://mrl.nyu.edu/~robbins/"/>
    <description>EMMET API Demo</description>
    <description kind="short">
      Extensible MultiModal Environment Toolkit API Demo
    </description>
    <offline-allowed/>
  </information>
  <security><all-permissions/></security>
  <resources>
    <j2se href="http://java.sun.com/products/autodl/j2se" version="1.4+" \
      initial-heap-size="64m" max-heap-size="192m"/>
    <jar href="EMMET.jar"/>
    <jar href="jars/hhresco.jar"/>
    <jar href="jars/jme.jar"/>
    <jar href="jars/lwjgl.jar"/>
    <jar href="jars/jmeui.jar"/>
    <jar href="jars/libsvm.jar"/>
    <jar href="jars/cgjsapi.jar"/>
    <jar href="jars/activation.jar"/>
    <jar href="jars/mailapi.jar"/>
    <jar href="jars/smtp.jar"/>
  </resources>
  <resources os="Windows">
    <nativelib href="lib/EMMET-native-win32.jar"/>
  </resources>
  <application-desc main-class="demos.EMMETGeometryPlacer"></application-desc>
</jnlp>

```

Figure 4.15: EMMET Geometry Placer Java™ WebStart JNLP File

and thus are also considered multimodal.

In conclusion, the EMMET Geometry Placer application verified EMMET's implementation of all proposed features. These features included support for: unimodal and multimodal user interface development utilizing less conventional input methods such as gesture and speech recognition; remote launching of EMMET created appli-

```
String results =  
    metInputHandler.getStatisticsCollector().getCommandCallStatistics();  
SMTPResultsMailer.sendResults(results);
```

Figure 4.16: EMMET Geometry Placer Remote Statistics Reporting

ation via the Web, and interface usage statistics gathering and remote reporting.

```

ADDEWOBJECT:
Summary:
MULTIMODAL: [total= 11; successes= 10 (90.9%); failures= 0 (0.0%); ignored= 1 (9.1%)]
SPEECH: [total= 0]
GESTURE: [total= 3; successes= 3 (100.0%); failures= 0 (0.0%); ignored= 0 (0.0%)]
MOUSE: [total= 11; successes= 6 (54.5%); failures= 0 (0.0%); ignored= 5 (45.5%)]
KEYBOARD: [total= 0]

Details:
(SUCCEEDED; MULTIMODAL(<addObject>, Add a green ball here); [SPEECH(<color>, green ), SPEECH(<object_type>, ball ),
SPEECH(<diectic_ref>, here ), MOUSE(<MOUSE_HOVER>, java.awt.Point[x=536,y=364], MOUSE_BUTTON_NONE),
SPEECH(<addObject>, Add a green ball here)]])
(SUCCEEDED; MULTIMODAL(<addObject>, Add a red ball here); [SPEECH(<color>, red ), SPEECH(<object_type>, ball ),
SPEECH(<diectic_ref>, here ), MOUSE(<MOUSE_HOVER>, java.awt.Point[x=302,y=480], MOUSE_BUTTON_NONE),
SPEECH(<addObject>, Add a red ball here)]])
(SUCCEEDED; MULTIMODAL(<addObject>, Add a yellow ball here); [SPEECH(<color>, yellow ), SPEECH(<object_type>, ball ),
SPEECH(<diectic_ref>, here ), MOUSE(<MOUSE_HOVER>, java.awt.Point[x=678,y=282], MOUSE_BUTTON_NONE),
SPEECH(<addObject>, Add a yellow ball here)]])
(SUCCEEDED; MULTIMODAL(<addObject>, Add a purple ball here); [SPEECH(<color>, purple ), SPEECH(<object_type>, ball ),
SPEECH(<diectic_ref>, here ), MOUSE(<MOUSE_HOVER>, java.awt.Point[x=524,y=570], MOUSE_BUTTON_NONE),
SPEECH(<addObject>, Add a purple ball here)]])
(SUCCEEDED; MULTIMODAL(<addObject>, Add a cube here); [SPEECH(<object_type>, cube ), SPEECH(<diectic_ref>, here ),
MOUSE(<MOUSE_HOVER>, java.awt.Point[x=824,y=388], MOUSE_BUTTON_NONE), SPEECH(<addObject>, Add a cube here)]])
(SUCCEEDED; MULTIMODAL(<addObject>, Add a pyramid here); [SPEECH(<object_type>, pyramid ), SPEECH(<diectic_ref>, here ),
MOUSE(<MOUSE_HOVER>, java.awt.Point[x=212,y=650], MOUSE_BUTTON_NONE), SPEECH(<addObject>, Add a pyramid here)]])
(SUCCEEDED; GESTURE(<ball>, 93.7%); [])
(SUCCEEDED; GESTURE(<pyramid>, 60.4%); [])
(SUCCEEDED; GESTURE(<cube>, 92.6%); [])
(SUCCEEDED; MULTIMODAL(<B>, KEY_DOWN); [MOUSE(<MOUSE_HOVER>, java.awt.Point[x=460,y=460], MOUSE_BUTTON_NONE), KEYBOARD(<B>, KEY_DOWN)]])
(SUCCEEDED; MULTIMODAL(<P>, KEY_DOWN); [MOUSE(<MOUSE_HOVER>, java.awt.Point[x=318,y=278], MOUSE_BUTTON_NONE), KEYBOARD(<P>, KEY_DOWN)]])
(SUCCEEDED; MULTIMODAL(<C>, KEY_DOWN); [MOUSE(<MOUSE_HOVER>, java.awt.Point[x=366,y=594], MOUSE_BUTTON_NONE), KEYBOARD(<C>, KEY_DOWN)]])
(SUCCEEDED; MULTIMODAL(<P>, KEY_DOWN); [MOUSE(<MOUSE_HOVER>, java.awt.Point[x=640,y=374], MOUSE_BUTTON_NONE), KEYBOARD(<P>, KEY_DOWN)]])
(SUCCEEDED; MOUSE(<MOUSE_DOWN>, java.awt.Point[x=238,y=346], MOUSE_BUTTON_2); [])
(SUCCEEDED; MOUSE(<MOUSE_DOWN>, java.awt.Point[x=604,y=456], MOUSE_BUTTON_2); [])
(SUCCEEDED; MOUSE(<MOUSE_DOWN>, java.awt.Point[x=104,y=522], MOUSE_BUTTON_2); [])
(SUCCEEDED; MOUSE(<MOUSE_DOWN>, java.awt.Point[x=556,y=612], MOUSE_BUTTON_2); [])
(SUCCEEDED; MOUSE(<MOUSE_DOWN>, java.awt.Point[x=454,y=244], MOUSE_BUTTON_2); [])
(SUCCEEDED; MOUSE(<MOUSE_DOWN>, java.awt.Point[x=328,y=468], MOUSE_BUTTON_2); [])

```

Figure 4.17: Geometry Placer “addNewObject” Command Usage Statistics

# Chapter 5

## Conclusion

The research work for this dissertation resulted in a number of contributions to the field of multimodal interfaces. These contributions arise from the implementation of a programmers' toolkit, called EMMET, that aids user interface designers in rapidly prototyping and evaluating speech and gesture based multimodal interfaces. Furthermore, the exhaustive explanation of EMMET's design and architecture provides a foundation upon which future researchers can build similar toolkits that may provide more functionality or support additional modalities.

The contributions resulting from the development and explanation of EMMET directly address Oviatt et al.'s summary of research challenges involved in advancing the field of multimodal interfaces [46]. Firstly, EMMET contributes to the field by providing a tool that facilitates the development of multimodal software. Secondly, EMMET contributes to the field by providing a method for collecting multimodal usage statistics to aid in the evaluation of alternative multimodal interface designs. Finally, the results obtained from such statistics provide insight and guidance for the design of future multimodal interfaces. Thus, as proposed, EMMET allows program-

mers to:

- explore speech and gesture based interface design without requiring an understanding of the details involved in the low-level implementation of speech or gesture recognition;
- quickly distribute these multimodal interface prototypes via the Web; and
- receive multimodal usage statistics collected remotely after each use of the programmer's application.

The application programmer interface provided by EMMET is consistent across all modalities and the details pertaining to the low-level implementation of each modality are hidden from the programmer. This consistency allows programmers who use EMMET to select which modality is best for a particular interaction, without this selection being influenced by any perceived or actual difficulties associated with using one modality versus another modality. Furthermore, this consistency facilitates programmers in quickly switching between the modalities used and, thus, it allows rapid prototyping and testing of alternative interface solutions.

EMMETs architecture is also highly modularized which allows the toolkit to be easily extended in order to support additional modalities. Examples of how to utilize this extensibility are presented throughout the discussion of EMMET's architecture, particularly in the section regarding the *controllers* package.

A command and trigger paradigm for writing multimodal interfaces is also introduced in EMMET. This paradigm moves the focus of user input handling to the interpretation of multimodal input events, rather than the input events themselves. Hence, programmers first write commands that encompass some functionality available to the user, and then identify which modal input events or combinations of modal

input events will trigger these commands. Thus, programmers can evaluate various command and trigger combinations without making changes to the commands driven by the occurrence of these triggers.

EMMET's support for speech triggers includes the ability to define tags within the grammar rules. These tags specify that certain words or phrases be automatically resolved, via multimodal integration, into representations that are meaningful to the application. For example, tagged deictic words are resolved into screen locations, and tagged words referring to objects in the 3D world are resolved into corresponding world objects.

The remote launching and remote usage statistics collection supported by EMMET's allows programmers to easily distribute their multimodal applications to an unlimited number of usability testers and receive nearly immediate feedback on how the application's interface is being used.

Future research and development possibilities include expanding EMMET to support additional modalities or to provide additional functionality within each modality. These additional modalities include but are not limited to eye tracking, 3D gesture recognition and lip feature recognition. Additional functionality includes: porting the tagging support provided for speech triggers to other modalities; augmenting statistics collection to provide more detailed data or mutual disambiguation statistics; automating the generation of JNLP scripts for launching EMMET applications; and implementing database server support for recording remotely collected statistics.

## Appendix A: Glossary of Terms

This glossary provides definitions for many of the Java and Object-Oriented Programming terms used throughout the explanation of EMMET's implementation and architecture. For a more complete glossary of such terms, please refer to glossary provided by David J. Barnes in his book, *Object-Oriented Programming with Java: An Introduction* [15]. Barnes' glossary is also available online at <http://www.cs.kent.ac.uk/people/staff/djb/book/glossary.html>.

**adapter class** A class in Java that implements an interface with a set of dummy or default methods. Allows one to subclass the adapter class and override just the methods needed.

**argument** In the definition of a method, arguments describe the data that will be passed as parameters. When the method is invoked, the data provided are called the parameters.

**array** A fixed length block of primitives or references

**callback method** A method defined in a class or interface designed to be overridden in a subclass or implementer to receive notification of events.

**constructor** The method or set of methods defined by a class to create new instances of that class.

**convenience method** A method defined in a class to provide access to information maintained by the class. Often, this same information can already be obtained, but in a more complicated and roundabout manner.

**derived class** Another term for a subclass. A class extends a base class, in order to derive a new class with all of the base class' variables and methods, plus some of its own.

**extend** Java terminology meaning to subclass. (eg. if class B is a subclass of class A, then class B extends class A)

**field** A class attribute. In other words, classes define fields that can be set to values in instances of that class.

**getter** A method that returns the value of a private or protected class field.

**instance** An instantiation of a class.

**list** A list of items which is theoretically unlimited in size.

**listener** An implementer of an interface or extender of an adapter class that overrides callbacks and is registered to receive notification of certain events.

**object** An instantiation of a class.

**parameter** A data item passed on the invocation of a method. In the definition of a method, the arguments describe the data to be passed as parameters.

**query method** A method defined in a class that returns information that is not otherwise accessible. The general difference between query methods and getters is that query methods connote information the must be generated or somehow derived from information maintained by the class, whereas getters just provide access to information already stored in existing fields.

**register** From the perspective of a listener implementation; to be added to a class in order to receive event notifications.

**singleton** A class design pattern in which only one instance of the class can exist at a time. Usually singletons are designed to be globally accessible.

**setter** A method that sets the value of a private or protected class field.

**subclass** A class which inherits from, or extends, a given class.

## Appendix B: UML Class Diagram Reference

The Unified Modeling Language (UML) class and package diagrams used extensively throughout the description of EMMET’s implementation follow the UML Version 2.0 specification as maintained by the Object Management Group (OMG) at <http://www.uml.org>. The one variation on this specification is with regard to the icons used in the class diagrams to indicate field and method properties. The following table is provided as a reference for interpreting these icons:

	default type (package visible)		constructor
	public type		abstract member
	default interface (package visible)		final member
	public interface		static member
			synchronized member
	default inner type (package visible)		type with public static void main(String[] args)
	private inner type		
	protected inner type		implements method from interface
	public inner type		overrides method from super class
	default inner interface (package visible)		
	private inner interface		
	protected inner interface		
	public inner interface		
	default field (package visible)		
	private field		
	protected field		
	public field		
	default method (package visible)		
	private method		
	protected method		
	public method		

# Bibliography

- [1] R. A. Bolt. 'Put-that-there': Voice and gestures at the graphics interface. In *Proceedings SIGGRAPH '80*, volume 14, pages 262–270, July 1980.
- [2] James H. Bradford. The human factors of speech-based interfaces: A research agenda. *ACM SIGCHI Bulletin*, 27(2):61–67, 1995.
- [3] Tom Brøndsted. Evaluation of recent speech grammar standardization efforts.
- [4] Jarir K. Chaar and M. J. Halliday. In-process evaluation for software inspection and test. Research Report RC 18630, IBM Corporation, 1993.
- [5] Adam Cheyer and Luc Julia. Multimodal maps: An agent-based approach, July 08 1995.
- [6] Chih chung Chang, Chih jen Lin, and Chih wei Hsu. A practical guide to support vector classification, October 29 2003.
- [7] J. D. Clarkson and J. Yi. Leathernet: A synthetic forces tactical training system for the usmc commander. In *Proceedings of the Sixth Conference on Computer Generated Forces and Behavioral Representation*, pages 275–281, Univ. of Central Florida, Orlando, 1996.
- [8] Jean claude Martin. TYCOON: Theoretical framework and software tools for multimodal interfaces, October 24 1998.
- [9] Joshua Clow and Sharon Oviatt. STAMP: A suite of tools for analyzing multimodal system processing, July 15 1998.
- [10] P. R. Cohen. The role of natural language in a multimodal interface. In *ACM UIST'92 Symp. on User Interface Software & Technology*, pages 143–149. 1992.
- [11] P. R. Cohen and S. L. Oviatt. The role of voice input for human-machine communication. August 25 2001.

- [12] Philip Cohen, David McGee, Sharon Oviatt, Lizhong Wu, Joshua Clow, Robert King, Simon Julier, and Lawrence Rosenblum. Projects in VR: Multimodal interaction for 2D and 3D environments. *IEEE Computer Graphics and Applications*, 19(4):10–13, July/August 1999.
- [13] Philip R. Cohen, Michael Johnston, David McGee, Sharon Oviatt, Jay Pittman, Ira Smith, Liang Chen, and Josh Clow. Quickset: multimodal interaction for distributed applications. *Proceedings of the ACM International Multimedia Conference and Exhibition 1997.*, pages 31–40, 1997.
- [14] Derek Coleman, Patrick Arnold, and Stephanie Bodiff. *Object-Oriented Development: The Fusion Method*. Prentice Hall, jul 1993.
- [15] David J. Barnes. *Object-Oriented Programming with Java: An Introduction*. Prentice-Hall, January 2000.
- [16] Sorin Dusan and James Flanagan. A system for multimodal dialogue and language acquisition. In *The 2nd Romanian Academy Conference on Speech Technology and Human-Computer Dialogue*, Bucharest, Romania, April 2003. Romanian Academy.
- [17] Jarkko Enden. Java speech API, January 11 2001.
- [18] Frans Flippo, Allen Krebs, and Ivan Marsic. A framework for rapid development of multimodal interfaces. In *Proceedings of the 5th International Conference on Multimodal Interfaces (ICMI-03)*, pages 109–116, New York, November 5–7 2003. ACM Press.
- [19] M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Object Technology Series. Addison Wesley, New York, 1997.
- [20] Micheal A. Grasso. *Speech Input in Multimodal Environments: Effects of Perceptual Structure on Speed, Accuracy, and Acceptance*. PhD thesis, University of Maryland, Baltimore, 1997.
- [21] S. Harada, J. Hwang, B. Lee, and M. Stone. “Put-That-There”: What, where, how? integrating speech and gesture in interactive workspaces. In *UBIHCISSYS 2003 Proceedings*, UbiComp 2003 Workshop 7, October 2003.
- [22] Hartwig Holzapfel, Kai Nickel, and Rainer Stiefelhagen. Implementation and evaluation of a constraint-based multimodal fusion system for speech and 3d pointing gestures. In *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces*, pages 175–182, New York, NY, USA, 2004. ACM Press.

- [23] Thomas G. Holzman. Computer-human interface solutions for emergency medical care. *interactions*, 6(3):13–24, 1999.
- [24] H. Hse and A.R. Newton. Graphic symbol recognition toolkit (hhreco) tutorial, 2003.
- [25] H. Hse and A.R. Newton. Sketched symbol recognition using zernike moments. In *Proceedings of the 17th International Conference on Pattern Recognition (ICPR'04)*, volume 1, pages 367–370, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California, Berkley, 2004. Pattern.
- [26] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. Addison Wesley, 2000.
- [27] Michael Johnston. Unification-based multimodal parsing, 1998.
- [28] Clare-Marie Karat, Christine Halverson, John Karat, and Daniel Horn. Patterns of entry and correction in large vocabulary continuous speech recognition systems. In *Proceedings of ACM CHI 99 Conference on Human Factors in Computing Systems*, volume 1 of *Speech and Multimodal Interfaces*, pages 568–575, 1999.
- [29] N. Krahnstoever, E.S Schapira, S. Kettebekov, and R. Sharma. Multimodal human-computer interaction for crisis management systems, 2003.
- [30] Joseph J. Laviola. MSVT: A virtual reality-based multimodal scientific visualization tool, November 27 1999.
- [31] C. Magerkurth, R. Stenzel, N. A. Streitz, and E. Neuhold. A multimodal interaction framework for pervasive game applications. In Antonio Krger and Rainer Malaka, editors, *Artificial Intelligence in Mobile Systems 2003 (AIMS 2003)*, pages 1–8, Seattle, USA, October 2003.
- [32] John Makhoul, Josh Bers, and Scott Miller. Designing conversational interfaces with multimodal interaction, April 03 1998.
- [33] Jack T. Marchewka and Tanya Goette. Implications of speech recognition technology. *Business Forum*, 17:26–30, Spring 1992.
- [34] D. Mcgee, I. Smith, J. Clow, M. Johnston, P. R. Cohen, and S. L. Oviatt. The efficiency of multimodal interaction: A case study, February 23 1998.
- [35] David Mcgee, Ira Smith, James A. Pittman, Michael Johnston, Philip R. Cohen, and Sharon L. Oviatt. Unification-based multimodal integration, 1997.

- [36] David Mcgee, Ira Smith, Jay Pittman, Josh Clow, Liang Chen, Michael Johnston, Philip R. Cohen, and Sharon Oviatt. Quickset: Multimodal interaction, April 07 2003.
- [37] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Inc, 2 edition, apr 1997.
- [38] J. G. Neal and S. C. Shapiro. Intelligent multi-media interface technology. In J. W Sullivan and S. W. Tyler, editors, *Proc. Architectures for Intelligent Interfaces: Elements and Prototypes*, pages 69–91, Lockheed AI Center, 1988. Superseded by [39].
- [39] Jeannette G. Neal and Stuart C. Shapiro. Intelligent multi-media interface technology. In Joseph W. Sullivan and Sherman W. Tyler, editors, *Intelligent User Interfaces*, pages 11–43. Addison Wesley, Reading, MA, 1991.
- [40] Mike O’Docherty. *Object-Oriented Analysis and Design: Understanding System Development with UML*. Wiley, John & Sons, Incorporated, may 2005.
- [41] Sharon Oviatt. Multimodal interfaces for dynamic interactive maps. In *Proceedings of ACM CHI 96 Conference on Human Factors in Computing Systems*, volume 1 of *PAPERS: Multi-Modal Applications*, pages 95–102, 1996.
- [42] Sharon Oviatt. Mutual disambiguation of recognition errors in a multimodal architecture. In *Proceedings of ACM CHI 99 Conference on Human Factors in Computing Systems*, volume 1 of *Speech and Multimodal Interfaces*, pages 576–583, 1999.
- [43] Sharon Oviatt. Ten myths of multimodal interaction. *Communications of the ACM*, 42(11):74–81, November 1999.
- [44] Sharon Oviatt. Multimodal system processing in mobile environments, July 31 2000.
- [45] Sharon Oviatt. Multimodal interfaces. In *Handbook of Human-Computer Interaction*. Lawrence Erlbaum, June 02 2003.
- [46] Sharon Oviatt, Phil Cohen, Lizhong Wu, Lisbeth Duncan, Bernhard Suhm, Josh Bers, Thomas Holzman, Terry Winograd, James Landay, Jim Larson, and David Ferro. Designing the user interface for multimodal speech and pen-based gesture applications: State-of-the-art systems and future research directions. *Human-Computer Interaction*, 15(4):263–322, 2000.

- [47] Sharon Oviatt, Antonella DeAngeli, and Karen Kuhn. Integration and synchronization of input modes during multimodal human-computer interaction. In *Proceedings of ACM CHI 97 Conference on Human Factors in Computing Systems*, volume 1 of *PAPERS: Speech, Haptic, & Multimodal Input*, pages 415–422, 1997.
- [48] Stepher R. Palmer and John M. Felsing. *A Practical Guide to Feature-Driven Development*. Pearson Education, nov 2001.
- [49] Hitesh Seth. Intoduction to SALT. *XML-Journal*, November 2002.
- [50] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, Saddle River, New Jersey, 1996.
- [51] Ben Shneiderman. *Designing the User Interface*. Addison Wesley Longman, third edition, 1998.
- [52] Ian Sommerville. *Software Engineering*. Addison Wesley, 7 edition, may 2004.
- [53] Stephen Stelting and Olav Maassen. *Applied Java Patterns*. The Sun Microsystems Press Java Series. Sun Microsystems Press, A Prentice Hall Title, Palo Alto, California, 2002.
- [54] Bernhard Suhm, Brad Myers, and Alex Waibel. Multimodal error correction for speech user interfaces. *ACM Transactions on Computer-Human Interaction*, 8(1):60–98, 2001.
- [55] Harald Trost, Michel Gnreux, and Ra Klein. A multimodal speech interface for accessing web pages, May 05 2000.
- [56] J. Vergo. A statistical approach to multimodal natural language interaction. In *Proceedings of the AAAI98 Workshop on Representations for Multimodal Human-Computer Interaction*, pages 81–85. AAAI Press, 1998.
- [57] Wolfgang Wahlster. User and discourse models for multimodal communication, October 30 1991.
- [58] Willie Walker, Paul Lamere, Philip Kwok, Bhiksha Raj, Rita Singh, Evandro Gouvea, Peter Wolf, and Joe Woelfel. Sphinx-4: A flexible open source framework for speech recognition. Technical Report SMLI TR2004-0811, Sun Microsystems Incorporated, 2004.
- [59] Bauhaus Universitaet Weimar, Doug A. Bowman, Ernst Kruijff, Ivan Poupyrev, and Joseph J. Laviola. An introduction to 3-D user interface design, July 02 2001.

- [60] Lizhong Wu, Philip R. Cohen, and Sharon L. Oviatt. Multimodal integration - A statistical view, December 13 1999.
- [61] Benfang Xiao, Cynthia Gir, and Sharon Oviatt. Multimodal integration patterns in children, July 08 2002.
- [62] M. Yeasin, N. Krahnstoever, R. Sharma, and S. Kettebekov. A real-time framework for natural multimodal interaction with large screen displays, October 29 2002.