Module Guide for EMgine: A Computational Model of Emotion for Enhancing Non-Player Character Believability in Games

Geneva M. Smith

October 15, 2022

Revision History

Date	Version	Notes
October 15, 2022	1.0	• Completed First Version based on SRS Version 1.0
July 21, 2022	0.8	• Module decomposition and definition based on SRS Version 0.5 • Missing references to Non-Functional Requirements in Section 5

Contents

List of Tables

List of Figures

1 Reference Material

This section records the units, symbols, abbreviations and acronyms, and mathematical notation that appears in this document for easy reference.

Abbreviations and Acronyms

Text	Description
AC	Anticipated Change
API	Application Programming Interface
DAG	Directed Acyclic Graph
M	Module
MG	Module Guide
NPC	Non-Player Character (Games)
OS	Operating System
SRS-R	Requirement from the Software Requirements Specification
SC	Scientific Computing
SRS	Software Requirements Specification
SRS-LC	Likely Change from the Software Requirements Specification
UC	Unlikely Change

2 Introduction

This is the Module Guide (MG) for the EMgine, a Computational Model of Emotion (CME) for Non-Player Characters (NPCs) to enhance their believability, with the goal of improving long-term player engagement. EMgine is for *emotion generation*, accepting user-defined information from a game environment to determines what emotion and intensity a NPC is "experiencing". How the emotion is expressed and what other effects it could have on game entities is left for game designers/developers to decide. For more information about EMgine and its requirement specification, see its Software Requirements Specification (SRS) document (Version 1.0).

2.1 Purpose of the Document

After completing an SRS, the MG is developed (Parnas et al., 1984). The MG specifies the modular structure of the system and is intended to allow both designers and maintainers to easily identify the parts of the software.

Decomposing a system into modules is a commonly accepted approach to developing software. A module is a work assignment for a programmer or programming team (Parnas et al., 1984). The module decomposition in this MG is based on the principle of information hiding (Parnas, 1972). This principle supports design for change, because the "secrets" that each module hides represent likely future changes. Design for change is valuable in software design, where modifications are frequent, especially during initial development as the solution space is explored. EMgine's design follows Parnas et al. (1984):

- System details that are likely to change independently should be the secrets of separate modules.
- Each data structure is implemented in only one module.
- Any other program that requires information stored in a module's data structures must obtain it by calling access programs belonging to that module.

2.2 Intended Readers of the Document

- **New Project Members**
  This document can be a guide for a new project member to aid their understanding of the overall structure and quickly find modules that are relevant to their work.
- **Maintainers**
  The MG's hierarchical structure improves the maintainers' understanding of the system when they need to make changes. It is crucial for a maintainer to update the relevant sections of the document after changes have been made.
- **Designers**
  Once the MG has been written, designers can use it to verify the system in various ways, such as consistency among modules, feasibility of the decomposition, and flexibility of the design.

2.3 Organization of the Document

The rest of the document is organized as follows:

- Section 3 lists the software requirements' anticipated/likely and unlikely changes

- Section 4 summarizes the module decomposition based on the anticipated changes
- Section 5 documents design decisions made to connect the software requirements to modules
- Section 6 gives a description for each module
- Section 7 includes two traceability matrices: one checks the completeness of the design against the requirements provided in the SRS, and the other shows the relation between anticipated changes and modules
- Section 8 describes the *uses* hierarchy between modules

3 Anticipated and Unlikely Changes

Possible changes are classified into two categories based on likeliness: Anticipated (Section 3.1) and Unlikely (Section 3.2).

3.1 Anticipated Changes

Anticipated changes are decisions and information that are to be hidden inside modules. Ideally, changing one of the anticipated changes will only require changing the one module that hides the associated decision. This is an adaptation of the *design for change* approach to software design. Where applicable, associated Likely Changes [SRS-LC] from the SRS (Ver. 1.0) appear in square brackets.

AC1: The implementation of time and, consequently, delta time.

AC2: The minimum value of emotion intensity [SRS-LC12].

AC3: The algorithm/method for calculating emotion intensity [SRS-LC4].

AC4: The algorithm/method for updating emotion intensity [SRS-LC14].

AC5: The algorithm/method for calculating intensity of *Surprise* [SRS-LC5].

AC6: The algorithm/method for calculating intensity of *Interest* [SRS-LC6].

AC7: The algorithm/method for calculating intensity of *Acceptance*.

AC8: The implementation of emotion kinds data structure [SRS-LC13].

AC9: The algorithm/method for generating emotion kinds [SRS-LC1, SRS-LC2, SRS-LC3].

AC10: The implementation of emotion state data structure.

AC11: The implementation of emotion state decay data structure.

AC12: The algorithm/method for decaying emotion intensity [SRS-LC7, SRS-LC8, SRS-LC9].

AC13: The algorithm/method for decaying an emotion state.

AC14: The implementation of emotion data structure.

AC15: The implementation of PAD point data structure.

AC16: The algorithm/method for converting an emotion state to a PAD point [SRS-LC10, SRS-LC11].

AC17: The implementation of world state data structure.

AC18: The implementation of world event data structure.

AC19: The implementation of distance between world states data structure.

AC20: The implementation of change in distance between world states data structure.

AC21: The implementation of goal data structure.

AC22: The implementation of plan data structure.

AC23: The definition and/or implementation of attention data structure.

AC24: The definition and/or implementation of social attachment data structure.

3.2 Unlikely Changes

Module design should be as general as possible, but a general system is typically more complex. Sometimes this complexity is not necessary and fixing some design decisions at the system architecture stage can simplify the software design. These decisions are *unlikely changes* because they are not intended to be changed due to the potentially large number of design elements that need to be modified to accommodate them.

EMgine does not have any known unlikely changes, as its goal is for users to choose, organize, modify, and reuse its modules as they see fit. These needs also motivate its conception as a library of components (see Section 5 for reasoning).

4 Module Hierarchy

This section provides an overview of the module design. Modules are summarized in a hierarchy decomposed by secrets (Table 1).The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented.

M1: Emotion Intensity Type Module

M2: Emotion Intensity Function Module

M3: Emotion State Type Module

M4: Emotion Generation Module

M5: Emotion Intensity Decay Rate Type Module

M6: Emotion Intensity Decay State Type Module

M7: Emotion Decay Function Module

M8: PAD Type Module

M9: PAD Function Module

M10: Emotion Type Module

M11: Emotion Function Module

M12: Goal Module

M13: Plan Module

M14: Attention Module

M15: Social Attachment Module

M16: Time Module

M17: World State Module

Level 1	Level 2	Level 3
Behaviour-Hiding Module	Emotion Intensity Module	[M1] Emotion Intensity Type Module
		[M2] Emotion Intensity Function Module
	Emotion State Module	[M3] Emotion State Type Module
		[M4] Emotion Generation Module
		[M5] Emotion Intensity Decay Rate Type Module
		[M6] Emotion Intensity Decay State Type Module
		[M7] Emotion Decay Function Module
	PAD Module	[M8] PAD Type Module
		[M9] PAD Function Module
	Emotion Module	[M10] Emotion Type Module
		[M11] Emotion Function Module
	[M12] Goal Module	–
	[M13] Plan Module	–
	[M14] Attention Module	–
	[M15] Social Attachment Module	–
Software Decision Module	[M16] Time Module	–
	[M17] World State Module	–

Table 1: Module Hierarchy

5 Connection Between Requirements and Design

The design of the system is intended to satisfy the requirements developed in the SRS (Version 1.0). In this stage, the system is decomposed into modules. Table 2 lists the connection between requirements and modules.

Library of Components This MG treats EMgine as a library of components due to its need to be an "understandable black box" design. EMgine must not require users to have an understanding of affective science and/or emotion research (SRS-NF17) and have reasonable compatibility with different agent architectures/frameworks (SRS-NF14), game entity embodiment (SRS-NF15), and development environments (SRS-NF13). Together with its requirement to clearly document usage information for user-facing (SRS-NF18), EMgine already lends itself to a black box as it focuses on the relation between inputs and outputs with no claims about the underlying process (Wehrle and Scherer, 1995, p. 601). This aligns with the description of domain-specific CMEs, where the transformation of inputs into affective phenomena is not important, as long as it has the desired effects (Hudlicka, 2019, p. 130–131; Osuna et al., 2020, p. 4–6). However, it would be difficult to support the Verifiability requirements (SRS-NF6, SRS-NF7) if one could not see how EMgine's parts passed information to each other. Therefore, EMgine's components should be black boxes, but not their connections so that its overall behaviour can be explained (Guimarães et al., 2022, p. 20).

This implies that a component-based software architecture (Qian et al., 2010, p. 248–261) is best, where each step in emotion generation is a component. This would:

- Increase EMgine's portability by specifying what each component is guaranteed to provide (Qian et al., 2010, p. 254) so that designers can use existing, tested and validated systems and enables fast and easy integration of new components (Rodríguez and Ramos, 2015, p. 443), increasing its compatibility with agent architectures/frameworks (SRS-NF14), game entity embodiment (SRS-NF15), and development environments (SRS-NF13)
- Allow designers to call EMgine's components as they see fit, supporting customization and modifiability (SRS-NF10, SRS-NF26), efficiency (SRS-NF4, SRS-NF5) (Carbone et al., 2020, p. 466), enabling users to specify how to use (SRS-NF21) and verify (SRS-NF6, SRS-NF7) EMgine outputs, and—potentially—demonstrating how EMgine could improve the player experience (SRS-NF27) and/or create novel game experiences (SRS-NF28)
- Have the potential for developing authoring tools that interface with and manage EMgine components, which could reduce total authorial effort (SRS-NF22)

Ongoing work on the FAtiMA architecture and its descendants, the FAtiMA Modular framework and FAtiMA Toolkit, found this approach successful (Mascarenhas et al., 2022, p. 8:2, 8:12–8:13). After gaining feedback from people in the games industry[1], the FAtiMA Toolkit's designers realized it as a library of components so that its parts can work autonomously. A library-based approach also alleviates some of EMgine's design requirements, as it no long has to "...be generic enough to encompass all possible forms of a perception-action cycle in an agent." (Mascarenhas et al., 2022, p. 8:12). This also increased the Toolkit's chances of adoption, as it allows game designers to use it in their existing systems and/or frameworks and avoiding the complexity and accessibility issues of other agent architectures (Guimarães et al., 2022, p. 3).

[1]As part of the "Realising an Applied Gaming Eco-system" (RAGE) project (https://cordis.europa.eu/project/id/644187).

In the same vein, EMgine would be a *library of components* so it can take advantage of these benefits. Users would also be free to focus on what sequence of components for emotion processing works for their needs because no one really knows what order they truly run in (Moffat, 1997, p. 142).

Pre-Built Components A library of components creates opportunities for EMgine to address additional nonfunctional requirements by pre-building some compound components, such as a default *"engine"*.

While each emotion generation component would be a black box (Qian et al., 2010, p. 253), a user might not know when they should use a component—or if it is even necessary, which might require some knowledge of affective science and/or emotion research to bridge (violates SRS-NF17). A component-based software architecture supports this need too by pre-building some compound components. It could come with an "engine"—a pre-built component that is itself a "system" of components (Qian et al., 2010, p. 249) that minimizes the necessary inputs for emotion generation. It would accept data and returns an emotion state, without the designer knowing how it works (Rodríguez and Ramos, 2015, p. 443). These would also serve as usage examples (SRS-NF18, SRS-NF19, SRS-NF17) and/or as parts for automating tasks (SRS-NF23).

In this form, EMgine would be similar to GAMYGDALA which compares itself to a physics engine (Popescu et al., 2014, p. 32). Designers have applied GAMYGDALA to: an arcade and puzzle game (Broekens, 2015); a narrative generation framework to drive character emotions (Kaptein and Broekens, 2015); implement affective decision-making in fighting game characters (Yuda et al., 2019); and modify it with the fuzzy logic, classical conditioning, and learning from another CME[2] (Code, 2015, p. 4). The breadth of games that GAMYGDALA appears in suggests that, even pre-packaged as a large component, EMgine could also find success.

Separating Data Types and Functions EMgine's module decomposition separates data types (M1, M5, M6, M8, M10) from functions that use them (M2, M7, M9, M11). While this increases the total modules, it also improves an end-user's ability to adopt individual parts of EMgine as needed (SRS-NF5, SRS-NF12, SRS-NF10) and change the underlying theories (SRS-NF9), and ensures that theory-agnostic components (e.g. M3) are separated from theory-specific ones (e.g. M4).

User-Defined APIs EMgine also encapsulates each Application Programming Interface (API) that end-users must define in their own modules (M16, M17, M14, M15). This allows a team to divide their implementation between members, reduces concerns regarding compatibility with othe EMgine components (SRS-NF10) and allows end-users to test and optimize them individually (SRS-NF5, SRS-NF7).

[2]FLAME (El-Nasr et al., 2000)

6 Module Decomposition

Modules are decomposed according to the principle of "information hiding" proposed by Parnas et al. (1984). Each module documents:

- Its *Secrets*, a brief statement of the design decision hidden by the module,
- Its *Services*, specifying *what* the module will do without documenting *how* to do it.
- A suggestion for what the module should be *Implemented By*. If this is *OS*, it means that the operating system or a standard programming language library provides the module. The system being designed does not need to provide them. If a there is a dash (–), the module is *not* a leaf node in the module hierarchy (Section 4). It is a "virtual" module, and will not have to be implemented unless required by a programming language.

6.1 Behaviour-Hiding Module

Secrets: The contents of the required behaviours.

Services: Includes programs that provide externally visible behaviour of the system as specified in the SRS (Ver. 1.0). The programs in this module will need to change if the SRS changes.

Implemented By: –

6.1.1 Emotion Intensity Module

Secrets: "Virtual" module containing the emotion intensity modules.

Services: Using Emotion Intensity data type, separated by data structure (M1) and methods (M2)

Implemented By: –

6.1.2 Emotion Intensity Type Module (M1)

Secrets: The internal representation of the Emotion Intensity data type.

Services: Stores the Emotion Intensity data type, enforces its constraints, and provides methods for using it.

Implemented By: EMgine

6.1.3 Emotion Intensity Function Module (M2)

Secrets: The method for calculating emotion intensity.

Services: Evaluates emotion intensity using Goal data, and optional Attention and Social Attachment data. Returns the result as Emotion Intensity data.

Implemented By: EMgine

6.1.4 Emotion State Module

Secrets: "Virtual" module containing the emotion state, emotion decay, and PAD modules.

Services: Using emotion state data type and associated emotion decay and PAD data types, separated by data structures (M3, M5, M8) and methods (M4, M7, M9).

Implemented By: –

6.1.5 Emotion State Type Module (M3)

Secrets: The internal representation of the Emotion State data type.

Services: Stores the Emotion State data type, enforces its constraints, and provides methods for using it.

Implemented By: EMgine

6.1.6 Emotion Generation Module (M4)

Secrets: The method for calculating which kind of emotion the current world state elicits.

Services: Evaluates what emotion an entity is "experiences" using Goal, Plan, and World State data. Returns the result as Emotion Kind data.

Implemented By: EMgine

6.1.7 Emotion Intensity Decay Rate Type Module (M5)

Secrets: The internal representation of the Emotion Intensity Decay Rate data type.

Services: Stores the Emotion Intensity Decay Rate data type, enforces its constraints, and provides methods for using it.

Implemented By: EMgine

6.1.8 Emotion Intensity Decay State Type Module (M6)

Secrets: The internal representation of the Emotion Intensity Decay State data type.

Services: Stores the Emotion Intensity Decay State data type, enforces its constraints, and provides methods for using it.

Implemented By: EMgine

6.1.9 Emotion Decay Function Module (M7)

Secrets: The method for calculating the temporal decay of emotion intensity.

Services: Evaluates emotion intensity using an Emotion Intensity data, Emotion Decay data, decay constants, and Time data. Returns the result as Emotion Intensity data.

Implemented By: EMgine

6.1.10 PAD Type Module (M8)

Secrets: The internal representation of the PAD data type.

Services: Stores the PAD data type, enforces its constraints, and provides methods for using it.

Implemented By: EMgine

6.1.11 PAD Function Module (M9)

Secrets: The method of converting Emotion State data into a PAD data.

Services: Evaluates "equivalent" PAD data from Emotion State data. Returns the result as PAD data.

Implemented By: EMgine

6.1.12 Emotion Module

Secrets: "Virtual" module containing the emotion modules.

Services: Using Emotion data type, separated by data structure (M10) and methods (M11)

Implemented By: –

6.1.13 Emotion Type Module (M10)

Secrets: The internal representation of the Emotion data type.

Services: Stores the Emotion data type, enforces its constraints, and provides methods for using it.

Implemented By: EMgine

6.1.14 Emotion Function Module (M11)

Secrets: The method for evaluating Emotion State data from Emotion data.

Services: Calculates Emotion-data dependant Emotion State data using a decay constant, and Emotion Decay and Time data. Returns the result as Emotion State.

Implemented By: EMgine

6.1.15 Goal Module (M12)

Secrets: Implementation of the Goal data structure.

Services: Stores the Goal data type, enforces its constraints, and provides methods for using it.

Implemented By: EMgine

6.1.16 Plan Module (M13)

Secrets: Implementation of the Plan data structure.

Services: Stores the Plan data type, enforces its constraints, and provides methods for using it.

Implemented By: EMgine

6.1.17 Attention Module (M14)

Secrets: Implementation of the Attention data structure.

Services: Provides the required data structures and methods associated with the Attention data type.

Implemented By: EMgine

6.1.18 Social Attachment Module (M15)

Secrets: Implementation of the Social Attachment data structure.

Services: Provides the required data structures and methods associated with the Social Attachment data type.

Implemented By: EMgine

6.2 Software Decision Module

Secrets: The design decision based on mathematical theorems, physical facts, or programming considerations. The secrets of this module are *not* described in the SRS. For EMgine, these are are user-defined modules.

Services: EMgine provides the API describing necessary data types and methods, but the end-user must implement them as they are unique to their application (e.g. video game).

Implemented By: –

6.2.1 Time Module (M16)

Secrets: Implementation of the Time and Delta Time data structures.

Services: Provides the required data structures and methods associated with the Time and Delta Time data types.

Implemented By: End-User

6.2.2 World State Module (M17)

Secrets: Implementation of the World State data structure.

Services: Provides the required data structures and methods associated with the World State data types.

Implemented By: End-User

7 Traceability Matrices

Two traceability matrices are provided to show connections between modules and the requirements developed in the SRS, and between modules and anticipated changes.

Module	Requirements
M1	SRS-R1, SRS-R3, SRS-R4, SRS-R19, SRS-R20
M2	SRS-R23
M3	SRS-R1, SRS-R6, SRS-R7, SRS-R19, SRS-R20, SRS-R21, SRS-R26, SRS-R27
M4	SRS-R22
M5	SRS-R1, SRS-R5
M6	SRS-R1, SRS-R8
M7	SRS-R24
M8	SRS-R1, SRS-R10
M9	SRS-R30
M10	SRS-R1, SRS-R9, SRS-R28, SRS-R29
M11	SRS-R25
M12	SRS-R1, SRS-R15
M13	SRS-R1, SRS-R16
M14	SRS-R1, SRS-R17
M15	SRS-R1, SRS-R18
M16	SRS-R1, SRS-R2
M17	SRS-R1, SRS-R11, SRS-R12, SRS-R13, SRS-R14

Table 2: Trace Between Requirements and Modules

Module	Anticipated Changes
M1	AC2
M2	AC3, AC5, AC6, AC7
M3	AC4, AC8, AC10
M4	AC9
M5	AC12
M6	AC11
M7	AC12
M8	AC15
M9	AC16
M10	AC14
M11	AC13
M12	AC21
M13	AC22
M14	AC23
M15	AC24
M16	AC1
M17	AC17, AC18, AC19, AC20

Table 3: Trace Between Anticipated Changes and Modules

Generally, each anticipated change should map to one module because it implies that each module contains information that only it knows about the design. However, in EMgine's design, three modules address multiple anticipated changes:

- M2 contains all of the functions that calculate emotion intensity. The SRS (Ver. 1.0) calculates the intensity for some emotion kinds differently. End-users do not need to know that they are calculated differently, and there is the potential for consolidating them into a single emotion intensity function. They also collectively share the same secret: the method for evaluating emotion intensity. Therefore, these are encapsulated into one module.

- M3 contains the types for emotion kinds and emotion state. The emotion state type (AC10) directly depends on the type of emotion kinds (AC8), and the function for updating emotion intensity (AC4) directly depends on the emotion state type. This tight coupling between components implies that changes to one likely mean changes to another, so they should be in the same module.

- M17 contains APIs for defining world states. These types are interrelated and inseparable, so they should be in the same module.

There is also one anticipated change, AC12 covered by two modules (M5, M7). This is due to their mutual reliance on the same underlying model. However, one could feasibly modify only one of these modules if that model does not change. Therefore, they are separated to afford users more flexibility in how they use them.

8 Use Hierarchy Between Modules

Parnas (1978) said of two programs, A and B, that A *uses* B if the correct execution of B might be necessary for A to complete the task as specified. That is, A *uses* B if there exist situations in which the correct functioning of A depends upon the availability of a correct implementation of B.

The *uses* hierarchy between modules (Figure 1) shows that the graph is a directed acyclic graph (DAG). Each level of the hierarchy offers a testable and usable subset of the system. Modules in the higher level of the hierarchy are simpler because they use modules from the lower levels.

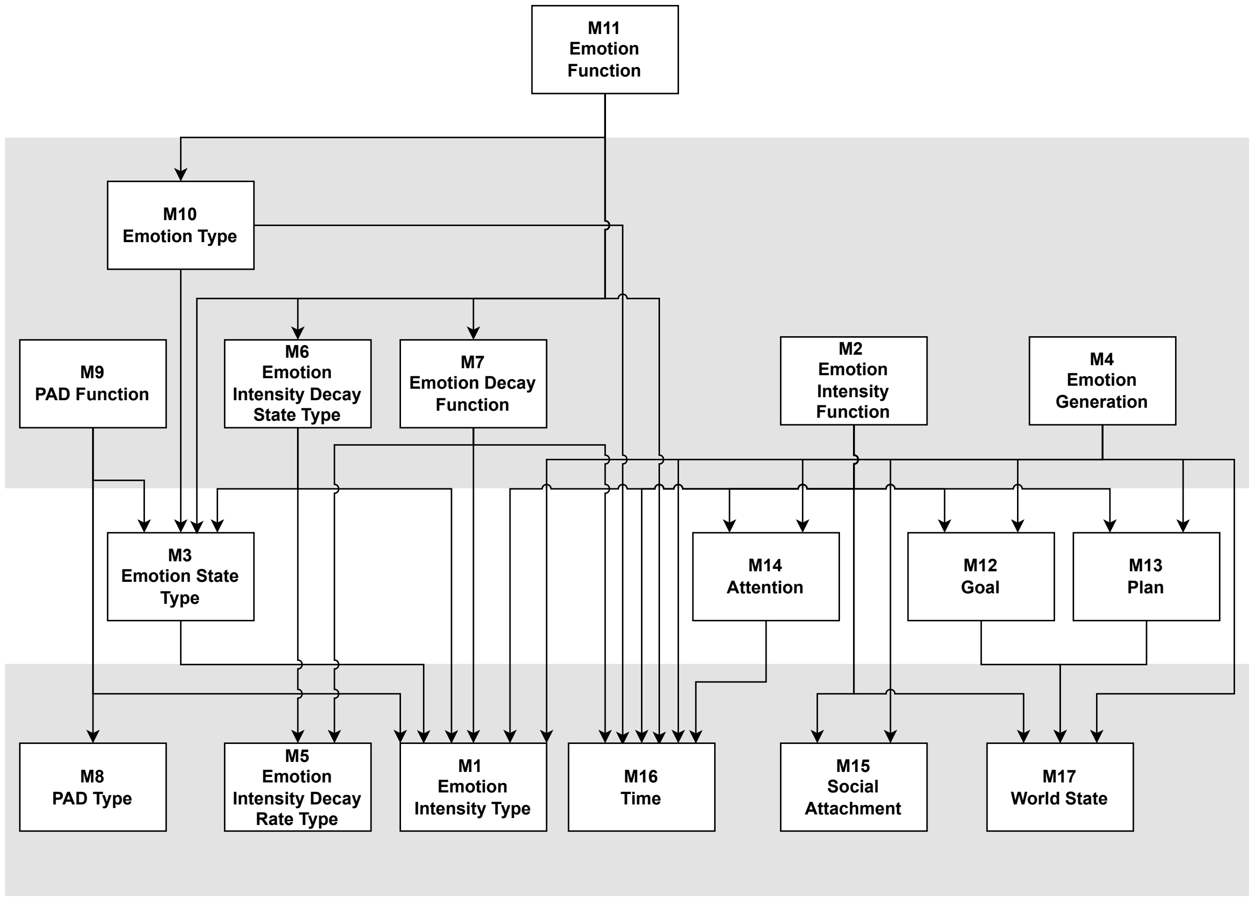

Figure 1: Use hierarchy among modules

9 References

Joost Broekens. 2015. Emotion Engines for Games in Practice: Two Case Studies using GAMYGDALA. In *2015 International Conference on Affective Computing and Intelligent Interaction (ACII 2015)*. September 21–24, 2015, Xi'an, China. IEEE, New York, NY, USA, 790–791.

John N. Carbone, James Crowder, and Ryan A. Carbone. 2020. Radically Simplifying Game Engines: AI Emotions & Game Self-Evolution. In *Proceedings of the 2020 International Conference on Computational Science and Computational Intelligence (CSCI)*. IEEE, Las Vegas, NV, USA, 464–472. https://doi.org/10.1109/CSCI51800.2020.00085

Douglas Code. 2015. Learning Emotions: A Software Engine for Simulating Realistic Emotion in Artificial Agents. Senior Independent Study Theses, Department of Computer Science, College of Wooster, Wooster, OH, USA, Paper 6543.

Magy Seif El-Nasr, John Yen, and Thomas R. Ioerger. 2000. FLAME—Fuzzy Logic Adaptive Model of Emotions. *Autonomous Agents and Multi-Agent Systems* 3, 3 (Sept. 2000), 219–257. https://doi.org/10.1023/A:1010030809960

Manuel Guimarães, Joana Campos, Pedro A. Santos, João Dias, and Rui Prada. 2022. Towards Explainable Social Agent Authoring tools: A case study on FAtiMA-Toolkit. arXiv:2206.03360 https://arxiv.org/abs/2206.03360

Eva Hudlicka. 2019. Modeling Cognition-Emotion Interactions in Symbolic Agent Architectures: Examples of Research and Applied Models. In *Cognitive Architectures*, Maria Isabel Aldinhas Ferreira, João Silva Sequeira, and Rodrigo Ventura (Eds.). Intelligent Systems, Control, and Automation: Science and Engineering, Vol. 94. Springer International Publishing, Cham, Switzerland, 129–143. https://doi.org/10.1007/978-3-319-97550-4_9

Frank Kaptein and Joost Broekens. 2015. The Affective Storyteller: Using Character Emotion to Influence Narrative Generation. In *Intelligent Virtual Agents (Lecture Notes in Computer Science, Vol. 9238)*, Willem-Paul Brinkmanand, Joost Broekens, and Dirk Heylen (Eds.). Springer International Publishing, Cham, Switzerland, 352–355.

Samuel Mascarenhas, Manuel Guimarães, Rui Prada, Pedro A. Santos, João Dias, and Ana Paiva. 2022. FAtiMA Toolkit: Toward an Accessible Tool for the Development of Socio-emotional Agents. *ACM Transactions on Interactive Intelligent Systems* 12, 1, Article 8 (March 2022), 30 pages. https://doi.org/10.1145/3510822

Dave Moffat. 1997. Personality Parameters and Programs. In *Creating Personalities for Synthetic Actors: Towards Autonomous Personality Agents*, Robert Trappl and Paolo Petta (Eds.). Lecture Notes in Computer Science, Vol. 1195. Springer Berlin Heidelberg, Berlin, Heidelberg, Germany, 120–165. https://doi.org/10.1007/BFb0030575

Enrique Osuna, Luis-Felipe Rodríguez, J. Octavio Gutierrez-Garcia, and Luis A. Castro. 2020. Development of Computational Models of Emotions: A Software Engineering Perspective. *Cognitive Systems Research* 60 (May 2020), 1–19. https://doi.org/10.1016/j.cogsys.2019.11.001

D. L. Parnas. 1972. On the Criteria To Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. https://doi.org/10.1145/361598.361623

David L. Parnas. 1978. Designing Software for Ease of Extension and Contraction. In *Proceedings of the 3rd International Conference on Software Engineering (ICSE'78)*. May 10–12, 1978, Atlanta, GA, USA. IEEE Press, Piscataway, NJ, USA, 264–277. https://doi.org/10.5555/800099.803218

David Lorge Parnas, Paul C. Clements, and David M. Weiss. 1984. The Modular Structure of Complex Systems. In *Proceedings of the 7th International Conference on Software Engineering (ICSE'84)*. March 26–29, 1984, Orlando, FL, USA. IEEE Press, Piscataway, NJ, USA, 408–417. https://doi.org/10.5555/800054.801999

Alexandru Popescu, Joost Broekens, and Maarten van Someren. 2014. GAMYGDALA: An Emotion Engine for Games. *IEEE Transactions on Affective Computing* 5, 1 (Jan.–March 2014), 32–44. https://doi.org/10.1109/T-AFFC.2013.24

Kai Qian, Xiang Fu, Lixin Tao, Chong-Wei Xu, and Jorge L. D'iaz-Herrera. 2010. *Software Architecture and Design Illuminated*. Jones and Bartlett Publishers, Sudbury, MA, USA. ISBN 978-0-7637-5420-4.

Luis-Felipe Rodríguez and Félix Ramos. 2015. Computational Models of Emotions for Autonomous Agents: Major Challenges. *Artificial Intelligence Review* 43, 3 (March 2015), 437–465. https://doi.org/10.1007/s10462-012-9380-9

Thomas Wehrle and Klaus R. Scherer. 1995. Potential Pitfalls in Computational Modelling of Appraisal Processes: A Reply to Chwelos and Oatley. *Cognition & Emotion* 9, 6 (Nov. 1995), 599–616.

Kaori Yuda, Maxim Mozgovoy, and Anna Danielewicz-Betz. 2019. Creating an Affective Fighting Game AI System with Gamygdala. In *2019 IEEE Conference on Games (CoG)*. IEEE, London, UK, 1–4.