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

Geneva M. Smith and Dr. Jacques Carette

Version 1.5 (March 23, 2023)

Revision History

Date	Version	Notes
March 23, 2023	1.5	• Updated to reflect SRS Version 1.5 • Collapsed Emotion Decay Function Module into Emotion Intensity Decay and Emotion State Decay Modules • Added description of EMgine's software architecture and component design
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 symbols, abbreviations, and acronyms that appear in this document for easy reference.

Table of Symbols

The table that follows summarizes the symbols used in this document along with their units, listed in alphabetical order. These correspond to the symbols in EMgine's Software Requirement Specification (SRS) at https://github.com/GenevaS/EMgine/blob/main/docs/SRS/EMgine_SRS.pdf

Symbol	Type	Description
$\mathbb{AT}_x$	SRS-TY17	The time elapsed while focusing on x
$\mathbb{D}$	SRS-TY12	The difference between two game world states
$\mathbb{D}_\Delta$	SRS-TY13	The change that a game world event makes to a game world state
$\mathbb{E}$	SRS-TY7	Emotion states over time
$\mathbb{ES}$	SRS-TY5	An emotion state
$\mathbb{ES}_\lambda$	SRS-TY6	An emotion decay state
$\mathbb{G}$	SRS-TY14	An entity's goal
$\mathbb{I}$	SRS-TY1	Emotion intensity
$\mathbb{I}_\Delta$	SRS-TY2	A change in emotion intensity
$\mathbb{I}_\lambda$	SRS-TY3	Emotion decay constant
$\mathbb{K}$	SRS-TY4	Allowable emotion labels/types
$\mathbb{P}$	SRS-TY15	An entity's plan
$P_{(P,A,D)}$	SRS-TY8	A point in PAD space
$\mathbb{S}$	SRS-TY10	A game world state
$\mathbb{S}_\Delta$	SRS-TY11	A game action that changes a game world state
$\mathbb{SA}$	SRS-TY16	A social attachment to entity A
$\mathbb{T}$	SRS-TY9	Abstract, linearly ordered time
$\mathbb{T}_\Delta$	SRS-TY9	The difference between two times

Abbreviations and Acronyms

Text	Description
AC	Anticipated Change
API	Application Programming Interface
CP	Component
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 1.5).

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 8 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 9 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. 1.5) appear in square brackets.

AC1: The minimum value of emotion intensity [SRS-LC11].

AC2: The algorithm/method for calculating emotion intensity [SRS-LC4].

AC3: The algorithm/method for updating emotion intensity with an intensity change.

AC4: The algorithm/method for calculating emotion intensity [SRS-LC5]. Although each of EMgine's supported emotion kinds have their own intensity model, end-users do not need to know that they are calculated differently. They also collectively share the same secret: their method for evaluating emotion intensity. Therefore, these are associated with one AC.

AC5: The algorithm/method for calculating intensity of *Acceptance*.

AC6: The implementation of emotion kinds data structure [SRS-LC12].

AC7: The algorithm/method for generating emotion kinds [SRS-LC1, SRS-LC2, SRS-LC3]. Although each of EMgine's supported emotion kinds have their own elicitation model, end-users do not need to know that they are calculated differently. They also collectively share the same secret: the method for emotion elicitation. Therefore, these are gathered into one module.

AC8: The implementation of emotion state data structure.

AC9: The implementation of emotion state decay data structure.

AC10: The algorithm/method for decaying emotion intensity [SRS-LC6, SRS-LC7, SRS-LC8].

AC11: The algorithm/method for decaying an emotion state.

AC12: The implementation of emotion data structure.

AC13: The implementation of PAD point data structure.

AC14: The algorithm/method for converting an emotion state to a PAD point [SRS-LC9, SRS-LC10].

AC15: The implementation of goal data structure.

AC16: The implementation of plan data structure.

AC17: The definition and/or implementation of social attachment data structure.

AC18: The definition and/or implementation of attention data structure.

AC19: The implementation of time and, consequently, delta time.

AC20: The implementation of World State View (WSV) data structure.

AC21: The implementation of world event data structure.

AC22: The implementation of distance between WSVs data structure.

AC23: The implementation of change in distance between WSVs 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 Module

M6: Emotion State Decay Module

M7: PAD Type Module

M8: PAD Function Module

M9: Emotion Type Module

M10: Emotion Function Module

M11: Goal Module

M12: Plan Module

M13: Social Attachment Module

M14: Attention Module

M15: Time Module

M16: 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
	Emotion Decay Module	**M5** Emotion Intensity Decay Module **M6** Emotion State Decay Module
	PAD Module	**M7** PAD Type Module **M8** PAD Function Module
	Emotion Module	**M9** Emotion Type Module **M10** Emotion Function Module
	Entity Module	**M11** Goal Module **M12** Plan Module **M13** Social Attachment Module **M14** Attention Module
Software Decision Module	World Module	**M15** Time Module **M16** 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 1.5). 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 suggests that a component-based software architecture (Qian et al., 2010, p. 248–261) is best, where each of EMgine's "tasks" is a discrete component that users can include, exclude, and change as needed. A component-based design approach is common in CME development (Osuna et al., 2021, p. 141). This approach would:

- Increase EMgine's portability by specifying what each component is guaranteed to provide 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)
- Mandates well-defined interfaces to show what each component requires and provides, including its configuration parameters, supporting customization (SRS-NF20, SRS-NF26)
- Allow designers to call EMgine's components as they see fit, supporting modifiability (SRS-NF10) and automation of emotion decay SRS-NF23
- Enable 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)
- Allow designers to add or swap EMgine's components, supporting *Allow the Integration of New Components* (RF4), *Allowing developers to choose which kinds of emotion EMgine produces* (RF5), and efficiency (SRS-NF4, SRS-NF5) (Carbone et al., 2020, p. 466)
- 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. 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). This also lets users focus on what emotion processing sequence works for their needs rather than constraining them to "EMgine's process" because no one really knows what order emotion processes 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 of EMgine's components are black-boxes (Qian et al., 2010, p. 253), users might not know when they should use a component—or if it is even necessary. This 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 allowing the specification of a prebuilt component—an "engine"—that is itself a "system" of components (Qian et al., 2010, p. 249) that minimizes the necessary decisions and inputs needed for emotion processing. It would accept data and return 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).

This prebuilt component or "engine" is similar to GAMYGDALA, which compares itself to a physics engine (Popescu et al., 2014, p. 32). Due to its plugin nature, designers have successfully applied GAMYGDALA to: arcade and puzzle games (Broekens, 2015); a narrative generation framework to drive character emotions (Kaptein and Broekens, 2015); and implement affective decision-making in fighting game characters (Yuda et al., 2019). A developer has also successfully integrated the fuzzy logic, classical conditioning, and learning from another CME[2] with GAMYGDALA (Code, 2015, p. 4). The breadth of games that developers have applied GAMYGDALA to and the potential for extensibility suggests that EMgine's inclusion of a large, prefabricated component could further increase its chances for success.

Separating Data Types and Functions Where possible, EMgine's module decomposition separates data types (M1, M7, M9) from functions that use them (M2, M8, M10). 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 (M15, M16, M14, M13). 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).

[1] As part of the "Realising an Applied Gaming Eco-system" (RAGE) project (https://cordis.europa.eu/project/id/644187).
[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. 1.5). 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. These are subdivided because several other modules need the Emotion Intensity data type and the methods for calculating emotion intensity are very likely to change.

Services: Using the Emotion Intensity data types and functions.

Implemented By: –

6.1.2 Emotion Intensity Type Module (M1)

Secrets: The internal representation of the Emotion Intensity and Emotion Intensity Change data types.

Services: Stores the Emotion Intensity and Emotion Intensity Change data types, enforce their constraints, and provides methods for manipulating them.

Implemented By: EMgine

6.1.3 Emotion Intensity Function Module (M2)

Secrets: The method for calculating emotion intensity.

Services: Evaluates emotion intensity using the Entity (M11, M12, M13, M14) and World (M15, M16) modules. Returns the result as an Emotion Intensity Change data type from the Emotion Intensity Type Module (M1).

Implemented By: EMgine

6.1.4 Emotion State Module

Secrets: "Virtual" module containing the Emotion State data type and generation modules. These are subdivided because the data type and generation modules rely on different affective theories.

Services: Using the Emotion State data type and emotion elicitation functions.

Implemented By: –

6.1.5 Emotion State Type Module (M3)

Secrets: The internal representation of the Emotion Kind and Emotion State data types.

Services: Stores the Emotion Kind and Emotion State data types, enforces their constraints, and provides methods for using them.

Implemented By: EMgine

6.1.6 Emotion Generation Module (M4)

Secrets: The method for evaluating if the current entity and world state elicit emotion(s).

Services: Evaluates what emotion an entity is "experiencing" using the Entity (M11, M12, M13, M14) and World (M15, M16) modules. Returns the result as tuples of Entity and World data types from the Entity and World modules.

Implemented By: EMgine

6.1.7 Emotion Decay Module

Secrets: "Virtual" module containing the emotion decay data types and function modules. The Emotion Intensity Decay and Emotion State Decay data types are separated into different modules because Emotion State Decay relies on more complex data types than Emotion Intensity Decay.

Services: Using the Emotion Decay data types and functions.

Implemented By: –

6.1.8 Emotion Intensity Decay Module (M5)

Secrets: The internal representation of the Emotion Intensity Decay Rate data type and the decay methods. The data type and methods are together because they are tightly coupled due to the underlying model.

Services: Stores the Emotion Intensity Decay Rate data type, enforces its constraints, and provides methods for manipulating it. Evaluates a "decayed" emotion intensity using that data type and the Delta Time data type from the Time module (M15), returned as an Emotion Intensity data type from the Emotion Intensity Type Module (M1).

Implemented By: EMgine

6.1.9 Emotion State Decay Module (M6)

Secrets: The internal representation of the Emotion Intensity Decay State data type and associated decay functions. These are separated from the Emotion Intensity Decay module because of their dependence on the Emotion State data type.

Services: Stores the Emotion Intensity Decay State data type, enforces its constraints, and provides methods for manipulating it. Evaluates a "decayed" emotion state using that data type and the Delta Time data type from the Time module (M15), returned as an Emotion State data type from the Emotion State Type Module (M3).

Implemented By: EMgine

6.1.10 PAD Module

Secrets: "Virtual" module containing the PAD data type and function modules. These are separated into different modules because the data type definition is highly unlikely to change whereas the conversion function is likely to change.

Services: Using the PAD data type and functions.

Implemented By: –

6.1.11 PAD Type Module (M7)

Secrets: The internal representation of the PAD data type.

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

Implemented By: EMgine

6.1.12 PAD Function Module (M8)

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

Services: Evaluates "equivalent" PAD representation of an Emotion State data type instance from the Emotion State Type module (M3). Returns the result as a PAD data type from the PAD Type Module (M7).

Implemented By: EMgine

6.1.13 Emotion Module

Secrets: "Virtual" module containing the Emotion data type and function modules. These are separated into different modules because the functions are for ease-of-use, which a user might not want to use because they have defined their own or they are not relevant to the task.

Services: Using the Emotion data type and functions.

Implemented By: –

6.1.14 Emotion Type Module (M9)

Secrets: The internal representation of the Emotion data type.

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

Implemented By: EMgine

6.1.15 Emotion Function Module (M10)

Secrets: The method for evaluating the next Emotion State from an Emotion data type.

Services: Calculates and adds the next Emotion State data type instance from the Emotion State Type Module (M3) to an Emotion data type instance using Emotion Intensity Decay State (M6) and Time modules (M15). Returns the result as an Emotion data type from the Emotion Type Module (M9).

Implemented By: EMgine

6.1.16 Entity Module

Secrets: "Virtual" module containing Entity-focused data types. Only the Goal module is mandatory and there are no dependencies between the Plan, Social Attachment, and Attention modules. Separating them into dedicated modules allows users to add and remove them as needed.

Services: Using the Goal, Plan, Social Attachment, and Attention data types.

Implemented By: –

6.1.17 Goal Module (M11)

Secrets: Implementation of the Goal data structure.

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

Implemented By: EMgine

6.1.18 Plan Module (M12)

Secrets: Implementation of the Plan data structure.

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

Implemented By: EMgine

6.1.19 Social Attachment Module (M13)

Secrets: Implementation of the Social Attachment data structure.

Services: Provides an API and default implementation for the Social Attachment data type.

Implemented By: EMgine

6.1.20 Attention Module (M14)

Secrets: Implementation of the Attention data structure.

Services: Provides an API and default implementation for the Attention 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 (M15)

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

Services: Provides an API of methods for using the Time and Delta Time data types necessary for EMgine.

Implemented By: End-User

6.2.2 World State Module (M16)

Secrets: Implementation of the World State data structure.

Services: Provides an API of methods for using the World State View (WSV), World Event, Distance Between WSVs, and Distance Change Between WSVs data types necessary for EMgine.

Implemented By: End-User

7 Module Composition into Components and Architecture

Module decomposition reduced EMgine's models into atomic units contained in "virtual" modules that are not implemented (Levels 1 and 2 in Table 1) but this is not necessarily how a user would visualize their groupings. For example, the hierarchy does not group the Emotion Generation and Emotion Intensity Function modules together despite their common dependencies (Figure 3). A user might similarly view them as a single unit because emotion intensity is only relevant if the associated emotion is present. Therefore, EMgine's assignment of the 16 modules to eight components aims to collect highly-related functionality into a comprehensive unit (Figure 1) that can be exchanged for another with comparable abilities while reducing inter-component connections (Figure 2):

CP1: Emotion Intensity collects the Emotion Intensity Type (**M1**) and dependant Emotion State Type (**M3**) modules because they represent EMgine's core emotion types. This also collects all EMgine-specific models necessary for users to define custom emotion kinds (SRS-R19, SRS-R20, SRS-R21). Since Emotion Intensity Type does not depend on other modules, this only reduces visible dependencies by one because the component hides Emotion State Type's dependency on Emotion Intensity Type.

CP2: Emotion Evaluation collects the Emotion Generation (**M4**) and Emotion Intensity Function (**M2**) modules due to the previously described interdependence of emotion generation and intensity calculations. Since these two modules require all the same inputs, grouping them as one unit also halves the visible dependencies on other modules.

CP3: Emotion Decay collects the Emotion Intensity Decay (**M5**) and Emotion State Decay (**M6**) because they capture the single "decay emotion" task. This hides Emotion State Decay's dependency on Emotion Intensity Decay while also halving the visible dependencies on other modules.

CP4: Emotion collects the Emotion Type (**M9**) and Emotion Function (**M10**) modules because EMgine intends the latter to be helper functions that make Emotion Type easier to use, effectively making Emotion Function wholly dependant on Emotion Type. This hides Emotion Function's dependency on Emotion Type while also halving the visible dependencies on other modules.

CP5: PAD collects the PAD Type (**M7**) and PAD Function (**M8**) modules because EMgine only requires a PAD data type to contain an emotion state that the PAD Function module has transformed into a PAD point. Since PAD Type does not depend on other modules, this only reduces visible dependencies by one as the component hides PAD Function's dependency on PAD Type.

CP6: Time contains only the Time (**M15**) module. It is separated from the World State module (**M16**) because there are significantly more dependencies on Time than World State, including ones that are otherwise independent of the world such as Emotion Decay (**M5** and **M6**). Time does not depend on other modules, so there are no visible dependency reductions.

CP7: World State contains only the World State (**M16**) module due to the need for Time to be its own component. World State does not depend on other modules, so there are no visible dependency reductions.

CP8: Entity contains the Goal (**M11**), Plan (**M12**), Social Attachment (**M13**), and Attention (**M14**) modules because of their common "entity representation" task. Unlike Time (**M15**) and World State (**M16**), there is no need to separate them into different components due to a higher number of dependencies on one module and not the others. They are also all relied on by the Emotion Evaluation

component (**C2**), so collecting the Entity modules together makes its use more convenient. There are no dependencies between Goal, Plan, Attention, or Social Attachment, so the component does not hide visible dependencies within itself. However, Goal and Plan's mutual dependency on World State (**M16**) is visible as a single connection rather than two separate ones.

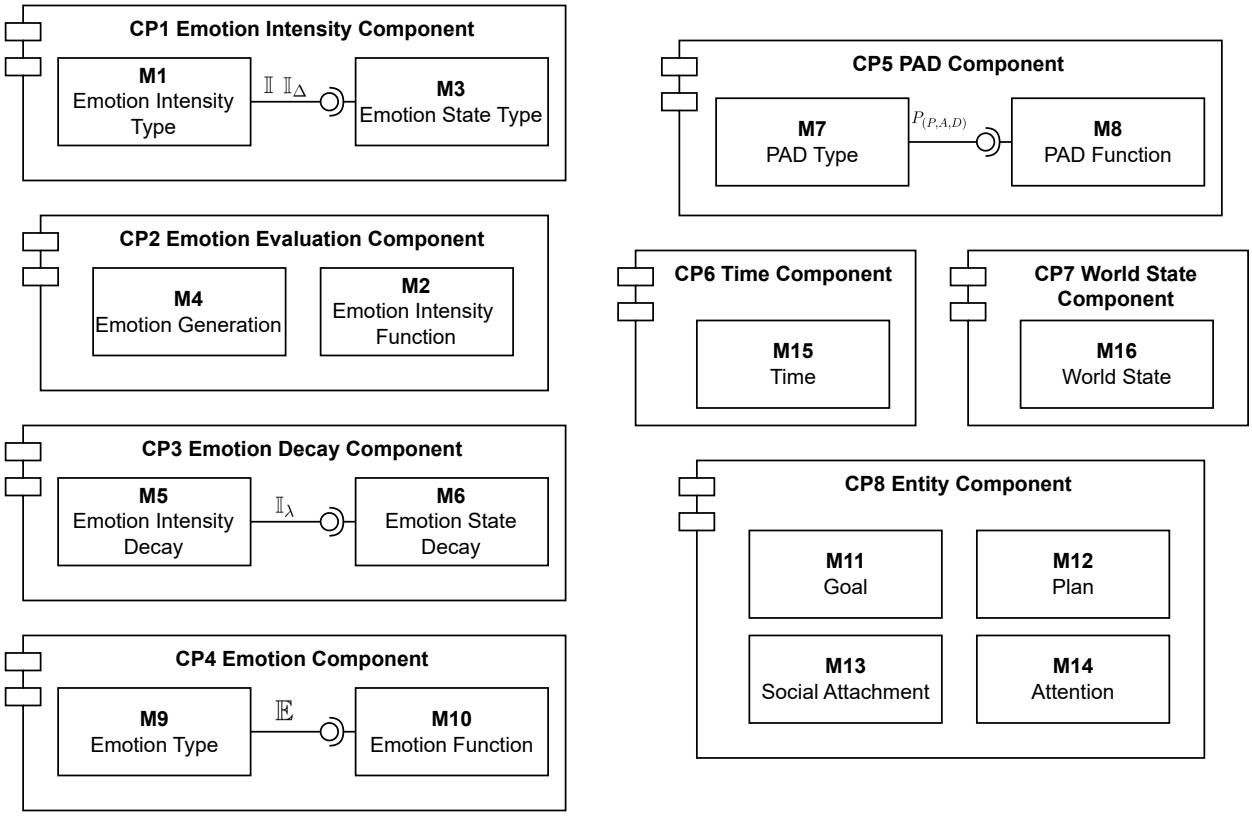

Figure 1: Organization of Modules in Components ("Ball" provides functionality that a "Cup" needs)

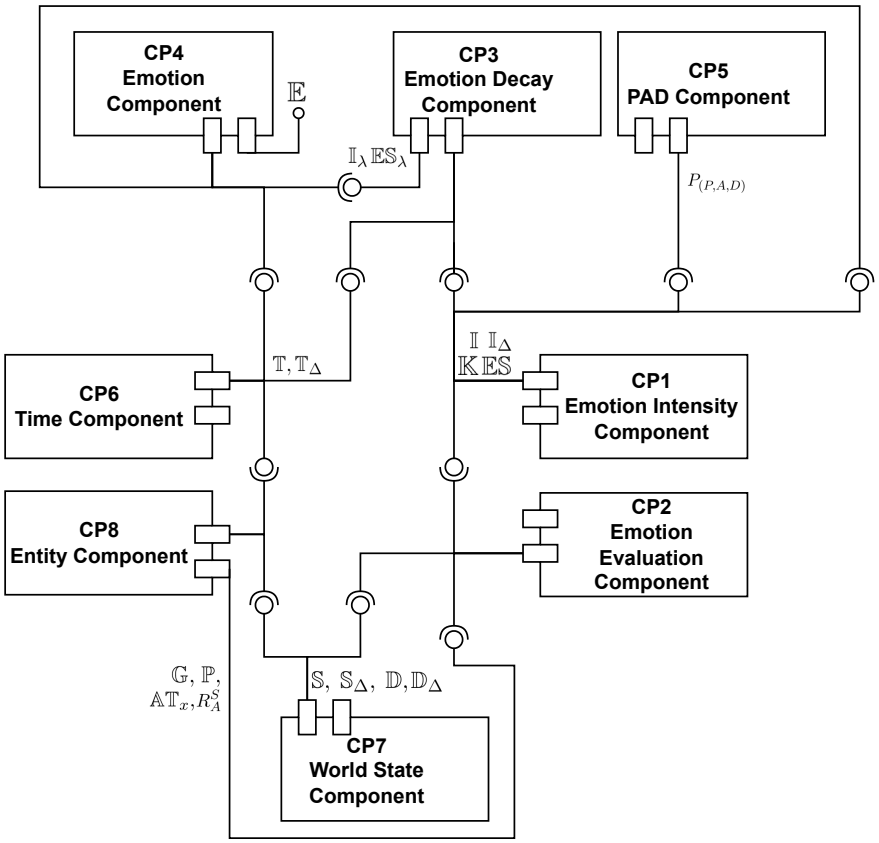

Figure 2: Relationships Between Modules ("Ball" provides functionality that a "Cup" needs)

8 Traceability Matrices

Traceability matrices show the connections modules and the requirements developed in the SRS, between modules and anticipated changes, and between components and modules.

Module	Requirements
M1	SRS-R1, SRS-R2, SRS-R3, SRS-R19, SRS-R20, SRS-R27
M2	SRS-R23
M3	SRS-R1, SRS-R5, SRS-R6, SRS-R19, SRS-R20, SRS-R21, SRS-R28
M4	SRS-R22
M5	SRS-R1, SRS-R4, SRS-R24
M6	SRS-R1, SRS-R7, SRS-R25
M7	SRS-R1, SRS-R9
M8	SRS-R31
M9	SRS-R1, SRS-R8, SRS-R29, SRS-R30
M10	SRS-R26
M11	SRS-R1, SRS-R15
M12	SRS-R1, SRS-R16
M13	SRS-R1, SRS-R17
M14	SRS-R1, SRS-R18
M15	SRS-R1, SRS-R10
M16	SRS-R1, SRS-R11, SRS-R12, SRS-R13, SRS-R14

Table 2: Trace Between Requirements and Modules

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

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, five modules address multiple anticipated changes:

- M1 collects the emotion intensity data types and function for updating an Emotion Intensity with an Emotion Intensity Change. Changing the minimum intensity (AC1) will likely cause changes to the method for "combining" the two intensity types (AC3). Therefore, these are gathered into one module.
- M3 contains the types for emotion kinds and emotion state. The emotion state type (AC8) directly depends on the type of emotion kinds (AC6), and the function for updating emotion intensity (AC3) directly depends on the emotion state type. This tight coupling between models implies that changes to one likely mean changes to another, so they should be in the same module.
- M6 contains the emotion decay state data type and the method for decaying an emotion state. The emotion state decay method (AC11) directly depends on the emotion state decay data type (AC9). This tight coupling between models implies that changes to one likely mean changes to another, so they should be in the same module.
- M16 contains APIs for defining world states. These types are interrelated and inseparable, so they should be in the same module.

9 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 3) 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.

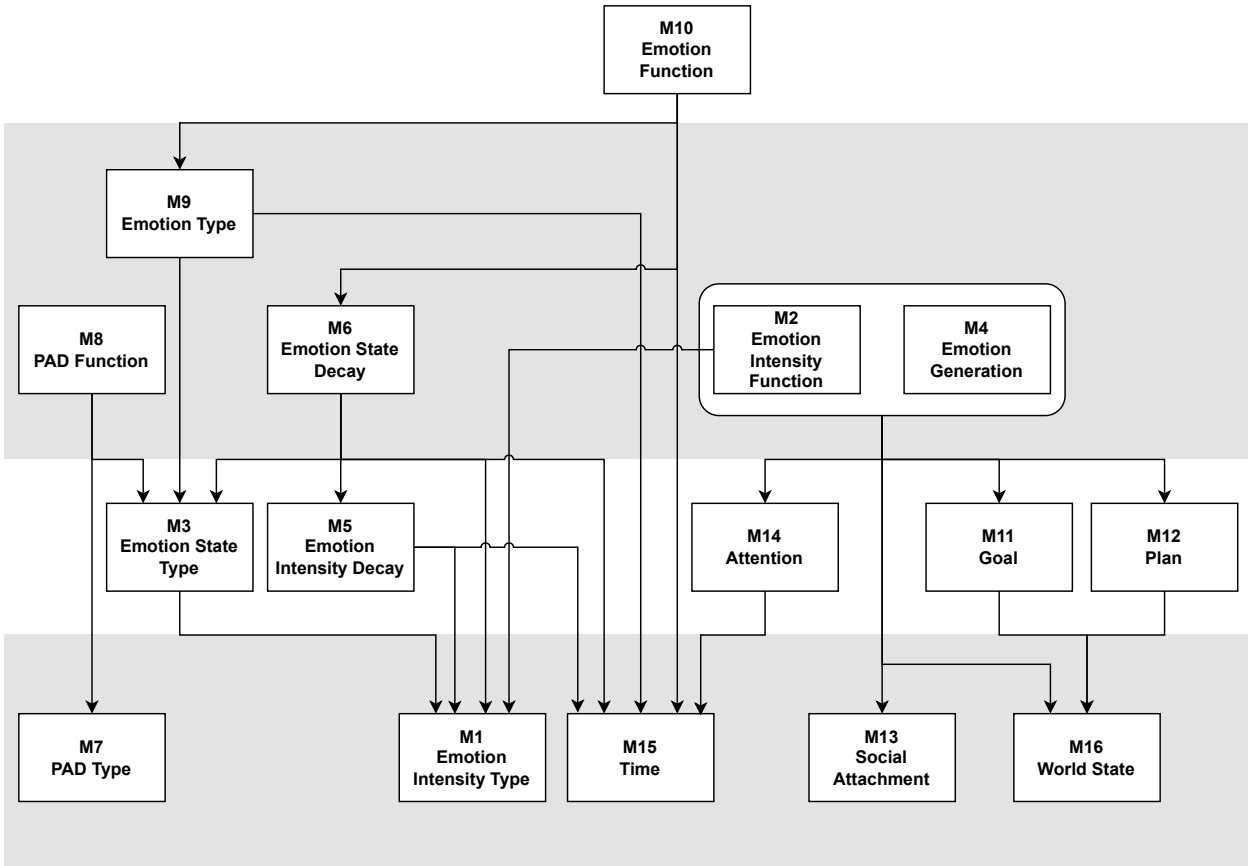

Figure 3: Use hierarchy among modules

10 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. https://doi.org/10.1109/ACII.2015.7344662

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. Advisor(s) Denise Byrnes. Paper 6543. https://openworks.wooster.edu/independentstudy/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. https://doi.org/10.1007/978-3-319-21996-7_38

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, and J. Octavio Gutierrez-Garcia. 2021. Toward Integrating Cognitive Components with Computational Models of Emotion Using Software Design Patterns. *Cognitive Systems Research* 65 (Jan. 2021), 138–150. https://doi.org/10.1016/j.cogsys.2020.10.004

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. https://doi.org/10.1080/02699939508408985

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. https://doi.org/10.1109/CIG.2019.8848031