Brilliant Students vs. Zombie Professors

Protocol Definition

Overview

This document defines the communication protocols for an interactive multi-player game, called the *Brilliant Students vs. Zombie Professors* (or *BSvZP)*. The system will consist of a variety of software components (processes) that will communicate with each other, namely: *Game (G), Playing Field (PF)*, *Clock Tower (CT)*, *Brilliant Students (BS)*, *Excuse Generators (EG)*, *Whining Spinners (WS)*, *Zombie Professors (ZP)*, a *Monitor (M)*, and *Referee* (*R*). The next section outlines the various types of conversations that may occur between these components and the general communication patterns that these conversations follow. It also defines messages that the protocols involves. The section after that defines how the software components must encode and decode them so they understand each other.

Conversations, Communication Patterns, and Messages

Table 1 lists the possible types of conversations involved in this system, along with which component initiates the conversation, other components involved, and general communication pattern. The communication pattern defines the possible message sequences in both normal and abnormal conditions.

The different communication patterns the protocols in Table 1 include *Limited Periodic Broadcast*, *One-way Send* and *Request-Reply*, each with various messages that come from the list of specialization of the *Request* and *Reply* class in Figure 1. Figures 2 - 9 illustrate the possible message sequences for the *One-way Send*, *Request-Reply*, and *3-party XYZ* patterns.

Table 1 – Conversations and Protocols for the *BSvZP*

(Note: protocols in gray will be implemented later.)

Protocol / Conversation	Initiator	Other Participants	Communication Pattern, Messages, and Semantics
Game Announcement *Use to let other component know about a newly created game that can be joined.*	*G*	*BS, EG, WS*	Limited Period Broadcast, with *GameAnnouncement* Message. NOTE: This communication will not be implemented for HW3. Real players will be manually given information about available games, who will then have to provide that information as configuration parameters to their

			agents
Join Game	*BS*, *EG*, *WS*, *ZP*, or *R*	*G*	*Request-Reply*, with *JoinGame* and AckNak as request and reply messages, where • The *AgentInfo* attrAgentibute of the *JoinGame* request contains a *ComponentInfo* object and that object with only the AgentType specified • If the *Status* of the *AckNak* message is *Success*, then the *ObjResult* in the *AckNak* message will be a completed *ComponentInfo* object. • If the *Status* of the *AckNak* message is *Failure*, the agent could not join the game for some reason and the *Message* of the *AckNak* contains the specific reason or error message.
Add Component	*G*	*PF*	*Request-Reply*, with *AddComponent* and *AckNak* messages, where • The *Component* attribute is the *AddComponent* is a *ComponentInfo* object that describes what needs to be added to the playing field. • If the *Status* of the *AckNak* message is *Success*, then the *IntResult* of the *AckNak* is the component's Id. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Remove Component	*R* *G*	*G* *PF*	*Request-Reply*, with *RemoveComponent* and *AckNak* messages • The *ComponentId* attribute is the *RemoverComponent* is the identifier of the component to remove from the playing field. • If the *Status* of the *AckNak* message is *Success*, then the *IntResult* of the *AckNak* is the component's Id. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Start Game	*G*	*PF, BS, EG, WS, ZP, M, R, or CT*	*One-way Send*, with *StartGame* as the messages. Later this may become a *Multicast.*
End Game	*R* *G*	*G* *PF, BS, EG, WS, ZP, M, or CT*	*One-way Send*, with *EndGame* as the messages. Later this may become a *Multicast.*
Get Configuration	*BS*, *EG*, *WS*, *ZP*, *M*, or *R*	*G*	*Request-Reply*, with *GetResource* and *AckNak* messages, where • The *GetType* in the *GetResource* message is *Game Configuration*. • If the *Status* of the *AckNak* message is *Success*, then the *ObjResult* of the *AckNak* is a *Configuration* object. • Otherwise, the request failed and the *Message* of the *AckNak*

			contains the specific reason or error message.
Get Playing Field Layout	*BS*, *EG*, *WS*, *ZP*, or *R*	*PF*	*Request-Reply*, with *GetResource* and AckNak messages, where • The *GetType* in the *GetResource* message is *Playing Field Layout*. • If the *Status* of the *AckNak* message is *Success*, then the *ObjResult* of the *AckNak* is a *PlayingFieldLayout* object. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Get Brilliant Student List	*BS*, *EG*, *WS*, *ZP*, or *R*	*PF*	*Request-Reply*, with *GetResource* and AckNak messages, where • The *GetType* in the *GetResource* message is *Brilliant Student List*. • If the *Status* of the *AckNak* message is *Success*, then the *ObjResult* of the *AckNak* is a *ComponentList* object containing *ComponentInfo* objects about all *BrilliantStudent* objects currently on the playing field. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Get Excuse Generator List	*BS*, *EG*, *WS*, *ZP*, or *R*	*PF*	*Request-Reply*, with *GetResource* and AckNak messages, where • The *GetType* in the *GetResource* message is *Excuse Generator List*. • If the *Status* of the *AckNak* message is *Success*, then the *ObjResult* of the *AckNak* is a *ComponentList* object containing *ComponentInfo* objects about all *ExcuseGenerator* objects currently on the playing field. Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Get Whining Spinner List	*BS*, *EG*, *WS*, *ZP*, or *R*	*PF*	*Request-Reply*, with *GetResource* and AckNak messages, where • The *GetType* in the *GetResource* message is *Excuse Generator List*. • If the *Status* of the *AckNak* message is *Success*, then the *ObjResult* of the *AckNak* is a *ComponentList* object containing *ComponentInfo* objects about all *WhiningSpinner* objects currently on the playing field. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Get Zombie Professor List	*BS*, *EG*, *WS*, *ZP*, or *R*	*PF*	*Request-Reply*, with *GetResource* and AckNak messages, where • The *GetType* in the *GetResource* message is *Excuse Generator List*. • If the *Status* of the *AckNak* message is *Success*, then the *ObjResult* of the *AckNak* is a *ComponentList* object containing *ComponentInfo*

			objects about all *ZombieProfessor* objects currently on the playing field. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Get Excuse	*BS*	*EG*	*Request-Reply*, with *GetResource* and AckNak messages, where • The *GetType* in the *GetResource* message is *Excuse*. • If the *Status* of the *AckNak* message is *Success*, then the *ObjResult* of the *AckNak* is an *Excuse* object. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Get Whining Twine	*BS*	*EG*	*Request-Reply*, with *GetResource* and AckNak messages, where • The *GetType* in the *GetResource* message is *Whining Twine*. • If the *Status* of the *AckNak* message is *Success*, then the *ObjResult* of the *AckNak* is an *Whining Twine* object. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Send Out Time Tick	*CT*	*BS*, *EG*, *WS*, or *ZP*	*One-way Sent*, with *TickMessage* as the messages. Later this may become a *Multicast.*
Validate Tick	*PF*	*CT*	*Request-Reply*, with *ValidateTick* and AckNak messages, where • The *ComponentId* attribute in *ValidateTick* message is the identify of the component that wants to use the *Tick* • If the *Status* of the *AckNak* message is *Success*, then the tick is valid. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Move	*BS* or *ZP*	*PF,* CT	*Request-Reply*, with *Move* and AckNak messages, where • The *ComponentId* attribute in the *Move* message is the identify of the component that wants to use the *Tick* • The *ToSquare* attribute in the *Move* message is where the agent (BS or ZP) wants to move • The *EnablingTick* attribute in the *Move* message is a valid Tick that agent hasn't used for any other purpose. • If the *Status* of the *AckNak* message is *Success*, then the move took place. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.

Throw Bomb	*BS*	*PF*	*Request-Reply*, with *Throw Bomb* and AckNak messages, where • The *ComponentId* attribute in the *Throw Bomb* message is the identify of the component that wants to throw the bomb. • The *Bomb* attribute in the *Throw Bomb* message has to be bomb containing at least one *Excuse* and one *Whining Twine* • The *TowardsSquare* attribute in the *Throw Bomb* message represent the target of the bomb. If the bomb doesn't have enough *Whining Twine* to go that distance, it will fail short, in some other square. • The *EnablingTick* attribute in the *Move* message is a valid Tick that agent hasn't used for any other purpose. • If the *Status* of the *AckNak* message is *Success*, then the bomb was thrown (but possibly not all the way to the target. The *ObjResult* attribute contains a *Square* object that describes where the bomb landed. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Eat	*ZP*	*PF*	*Request-Reply*, with *Eat* and *AckNak* messages, where • The *ZombieId* attribute in the Eat message is the identity of the zombie that wants to eat something else. • The *TargetId* attribute is the identity of the target agent that the zombie wants to eat. • If the *Status* of the *AckNak* message is *Success*, then the *Eating* took place. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Change Strength	*PF*	*BS*, *EG*, *WS*, or *ZP*	*Request-Reply*, with *ChangeStrength* and *AckNak* messages, where • The *DeltaValue* attribute is the delta value that needs to be apply to the receiving agent's current strength. • If the *Status* of the *AckNak* message is *Success*, then the operation was successful. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
Collaborate	*BS*	*BS*	*Request-Reply*, with *Collaborate* and *AckNak* messages, where • If the *Status* of the *AckNak* message is *Success*, then the *ObjResult*

			attribute contains *ComponentInfo* object that describes the current target of the receiving agent. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.
GetStatus	*M* or *R*	*BS, EG, WP,* or *ZB*	*Request-Reply*, with *GetStatus* and *AckNak* messages, where • If the *Status* of the *AckNak* message is *Success*, then the *ObjResult* attribute contains *StatusInfo* object that describes the current status of the receiving agent. • Otherwise, the request failed and the *Message* of the *AckNak* contains the specific reason or error message.

Message Encoding / Decoding

A message will be encoded recursively using the following rules:

1. The encoding of a *Message* object involves writing its Class Id, the length of its encoded properties, and its properties into a *ByteList*.

1.1. The encoding properties process is a pre-defined order of the class

1.2. Each property is encoded as follows:

1.2.1. A primitive numeric value (e.g. an integer) is written out in network byte order

1.2.1.1. Byte – 1 byte

1.2.1.2. Int16 – 2 bytes

1.2.1.3. Int32 – 4 bytes

1.2.1.4. Int64 – 8 bytes

1.2.1.5. Single Precision Real – 4 bytes

1.2.1.6. Double Precision Real – 4 bytes

1.2.2. A char is encoded by writing a two-byte Unique representation of the char value.

1.2.3. A string is encoded by writing out its length as an Int16 (in network byte order) and a sequence of bytes, where the bytes are a Unicode representation of the string.

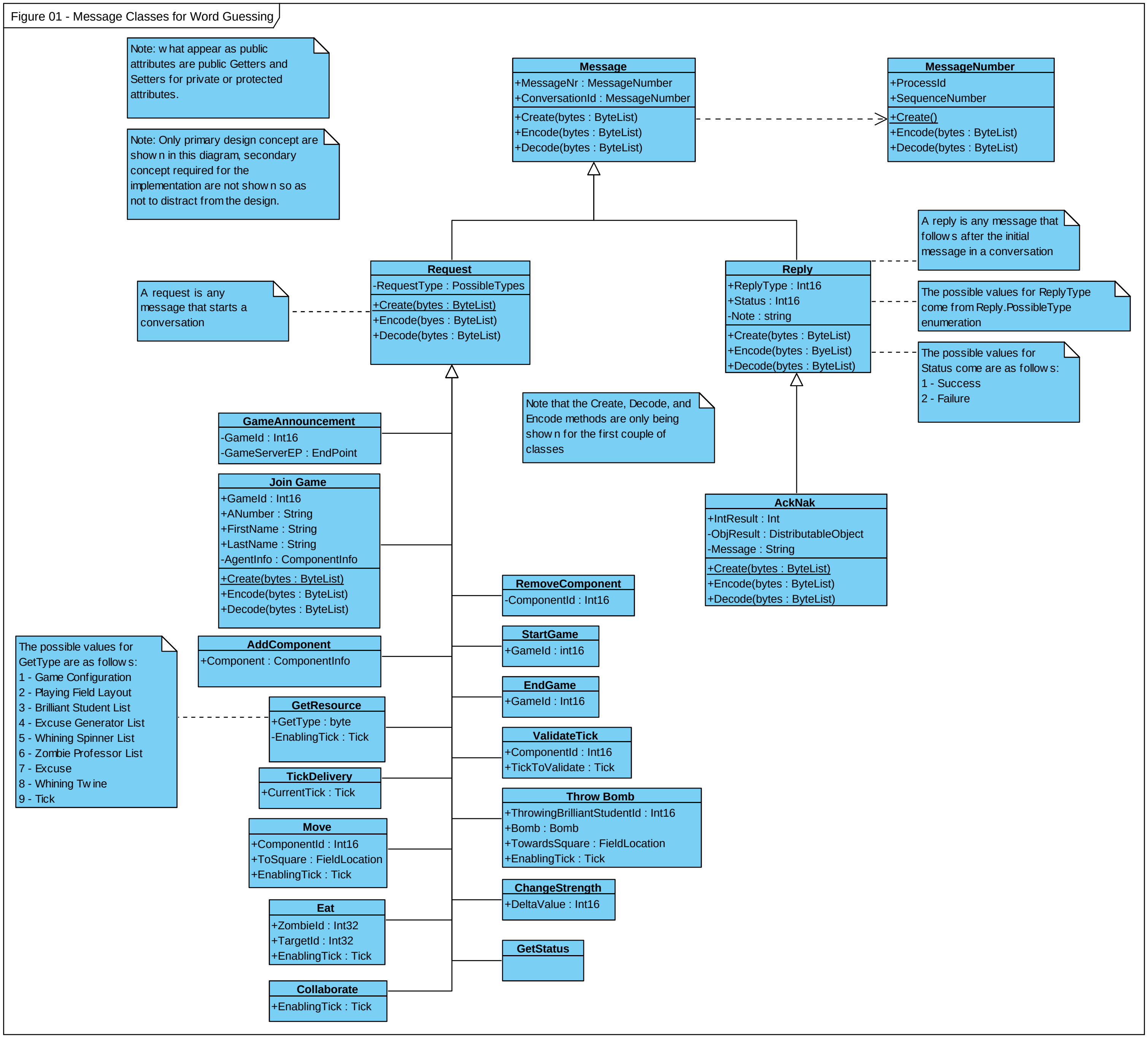
1.2.4. A Boolean value is written out as a byte with a value of 0 (false) or 1 (true)

1.2.5. An array or list of primitive values is encoded by first writing out the count of elements in the array or list as an Int16 (in network byte order), followed by an encoding of each value following rules 1.2.1 – 1.2.4

1.2.6. A property whose value is object is first represented from a byte containing a “1” for True or a “0” for False. A true means that the object is present and its encoding follows. A false means the object is not present. The encoding of the objects follows Rule 1 recursively.

1.2.7. An array or list of objects is encoded by first writing out the count of elements in the array or list as an Int16 (in network byte order), followed by an encoding of each object following Rule 1

Figure 01 - Message Classes for Word Guessing



Note: what appear as public attributes are public Getters and Setters for private or protected attributes.

Note: Only primary design concept are shown in this diagram, secondary concept required for the implementation are not shown so as not to distract from the design.

A request is any message that starts a conversation

A reply is any message that follows after the initial message in a conversation

The possible values for ReplyType come from Reply.PossibleType enumeration

The possible values for Status come are as follows:
1 - Success
2 - Failure

Note that the Create, Decode, and Encode methods are only being shown for the first couple of classes

The possible values for GetType are as follows:
1 - Game Configuration
2 - Playing Field Layout
3 - Brilliant Student List
4 - Excuse Generator List
5 - Whining Spinner List
6 - Zombie Professor List
7 - Excuse
8 - Whining Twine
9 - Tick

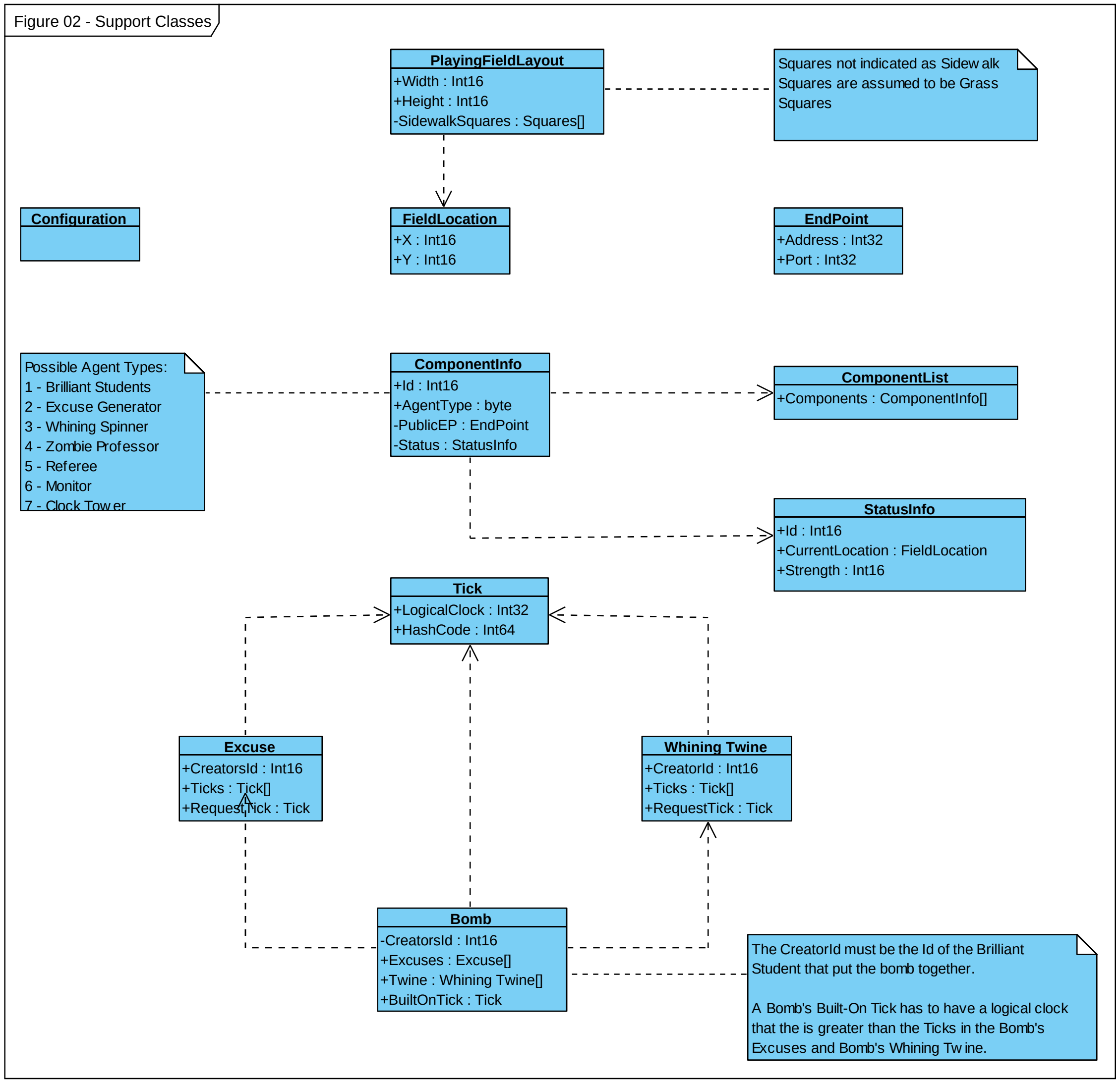


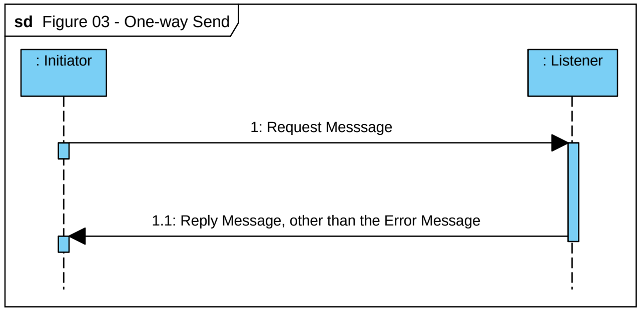
Figure 02 - Support Classes

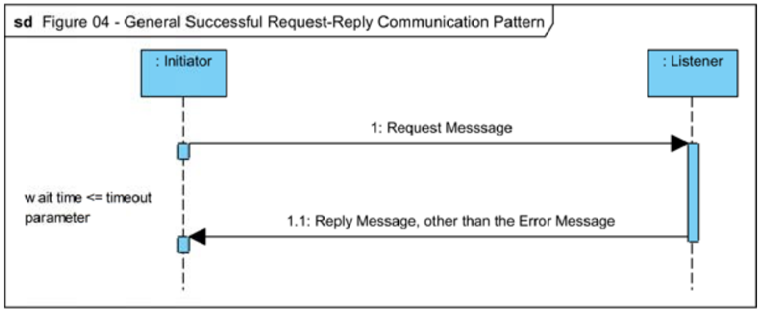
Squares not indicated as Sidewalk Squares are assumed to be Grass Squares

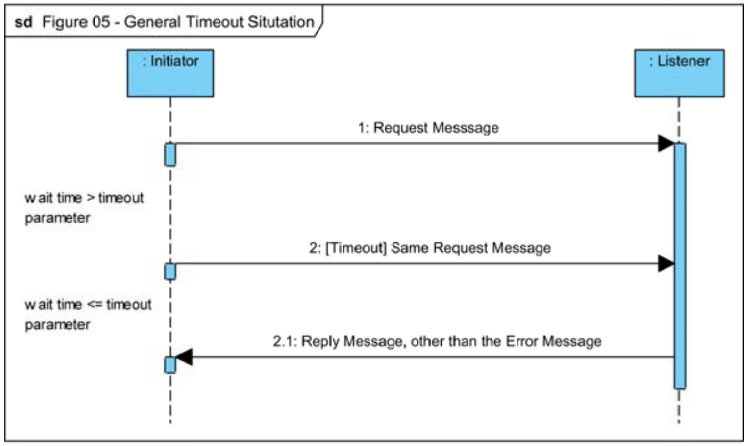
Possible Agent Types:
1 - Brilliant Students
2 - Excuse Generator
3 - Whining Spinner
4 - Zombie Professor
5 - Referee
6 - Monitor
7 - Clock Tower

The CreatorId must be the Id of the Brilliant Student that put the bomb together.

A Bomb's Built-On Tick has to have a logical clock that the is greater than the Ticks in the Bomb's Excuses and Bomb's Whining Twine.
]

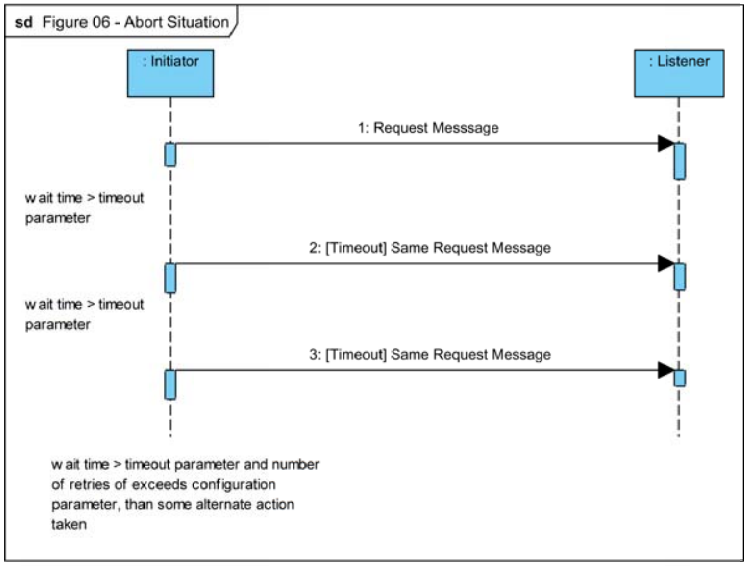

sd Figure 03 - One-way Send
: Initiator
: Listener
1: Request Messsage
1.1: Reply Message, other than the Error Message


sd Figure 04 - General Successful Request-Reply Communication Pattern
: Initiator
: Listener
1: Request Messsage
w ait time <= timeout parameter
1.1: Reply Message, other than the Error Message



sd Figure 05 - General Timeout Situtation

: Initiator
: Listener
1: Request Messsage
wait time > timeout parameter
2: [Timeout] Same Request Message
wait time <= timeout parameter
2.1: Reply Message, other than the Error Message


sd Figure 06 - Abort Situation
: Initiator
: Listener
1: Request Messsage
wait time > timeout parameter
2: [Timeout] Same Request Message
wait time > timeout parameter
3: [Timeout] Same Request Message
wait time > timeout parameter and number of retries of exceeds configuration parameter, than some alternate action taken