**WarGames: A Representation of Organized Collaboration in Combat Artificial Intelligence**

Shane Nalezyty

Blackburn College

**ABSTRACT**

WarGames is a modern combat simulator in which two teams compete against one another on a battlefield to achieve victory. Each team is led by a team leader who continually receives information from his soldiers and adjusts their strategies accordingly. The first team to eliminate all opposing troops or reach the enemy base is declared the victor.

Using the Unity3D [13] game engine and a third-party codebase Tactical Shooter AI, I build a system of artificial intelligence containing multiple communicating agents working cooperatively. Teams consist of a single team leader and three soldiers under the Team leader's’ command. To accomplish this, I design and implement a system of goals team leaders can distribute to soldiers who work to satisfy those goals.

Goals give soldiers direction in the battlefieldand instruct soldiers on how aggressive they should be while navigating the battlefield. Using these goals, team leader’s work towards objectives that win the game for the team. A communication network provided to each agent. Agents of the same team share a single communication network. The communication network allows for team leaders to issues goals and requests to soldiers by sending soldiers messages over the communication network. Soldiers can respond to messages received from their team leader by sending messages containing battlefield information they perceive.

Team leaders contain a facility for the storage of information about the battlefield. Information stored inside the team leader’s storage facility is gathered from the team leader’s sensors on the environment and from messages sent from the soldiers containing knowledge the sending soldier currently perceives. If a team leader dies, then the knowledge gathered is destroyed. When a soldier becomes the team leader to replace the lost leader his knowledge base is empty, and all information gained previously must be reacquired.

A method for allowing agents to find cover dynamically is also defined. The codebase I am building from, Tactical Shooter AI [12], includes several methods for manually assigning cover and a single method of dynamic cover discovery. My method is an extension of the original Tactical Shooter AI dynamic cover discovery method. I expand on the original design by allowing for progressive cover to be created more easily, adjust how long agents use and find cover based on aggression, and ensure agents take cover on the proper side of the map.

# INTRODUCTION

When attempting to simulate an organized team structure in a combat scenario, I encountered many interesting problems which included... Multi-agent systems require precise cooperation when attempting to achieve the same objective. Due to this being an attempt to simulate a three-dimensional combat environment each agent has its own eyes in the form of sensors that provide it with environmental information from the agent’s perspective. For that reason, each agent’s collected information will most likely differ.  The synergy mandatory in multi-agent systems requires the sharing of perspective information between agents.

Combat throughout the ages has almost always contained a clear command hierarchy. This structure is no different in modern combat scenarios. In modern combat, small groups of soldiers commanded by one controlling soldier are normally used. In the US Army, this small group is known as a fireteam. In the US Army Field Manual, a fireteam consists of a single team leader, rifleman, grenadier, and automatic rifleman. A team leader is responsible for the tactical leadership of his squad as such [8], WarGames must include a system of command to create a command hierarchy within agent teams.

The method in which agents communicate must be built to match a logical setup about the command hierarchy. Standard soldiers should be able to receive commands and requests for information while only sending battlefield information to the team leader. Likewise, team leaders should never receive commands, but must be able to interoperate and store the information received from soldiers.

Most combat artificial intelligence is designed for games or military battlefield simulators [3]. Often enough military simulators are the only systems containing commanded team-based artificial intelligence. Games containing team based combat artificial intelligence implementations tend to be closed source systems designed strictly for the use by the company that built them. Expanding on current publically accessible combat AI creates opportunities for the advancement of AI in games and simulators developed by smaller groups of or independent developers. Open source development provides fast deployment of software, the flexibility of use, and advancement of knowledge for developers in need. Providing an advanced open combat AI system will likely contribute to independent development.

Teams of organized, intelligent agents also create more difficult scenarios for players of both games and simulations. Simply including both an independent and team-based multi-agent systems can allow the designer to establish a larger variety of complexity for players to experience.

These systems are hardly developed for smaller projects because of the overhead required. Often enough a simple AI solution will suffice despite being the optimal solution. Combat AI Systems of this nature require the development of a communication architecture and the creation of at least two agents, of which the most complex being the commanding agent. A simpler solution may only require a single agent.

Commanding agents are very intricate. Team leaders have the responsibility to keep track of any sub-agents, provide those agents orders to follow, and receive and store information to sub-agents report for later use. On top of the infrastructure allowing this to happen, commanding agents need to direct sub-agents in a way that responds quickly and accurately to the changing environment. Selecting when to change what goals agents are executing, what formation agents should use to navigate the battlefield, how to respond to the enemy presence, and when to prioritize objectives over others are all difficult questions a commanding agent must be able to answer. Development projects on tight budgets may consider the time required to design, implement, and debug such sophisticated systems as better spent on other portions of their development.

Also, AI algorithms must be generalized. All agents in the system should be able to adapt to various, but structurally similar, environments. Without overcoming the initial obstacle of generalizing the operations of the agents, a system of this complexity would require a vast amount of configuration on a per-scenario basis and would lead to an increase of development time.

## Key Components

These key terms important to the fundamental understand of the paper. A comprehensive list of WarGames’ nomenclature is available in the glossary provided with this document.

**Agent** – WarGames agents follow the same definition of agents and reactive agents described by Stan Franklin and Art Graesser in their research: "An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future." [1]

**Multi-agent systems** – Gerhard Weiss also provides the definition for multi-agent systems: “A multi-agents system is a system that contains a set of agents that interact with communications protocols and can act on their environment. Different agents have different spheres of influence, in the sense that they have control (or at least can influence) on different parts of the environment. These spheres of influence may overlap in some cases; the fact that they coincide may cause dependencies reports between agents” [15].

**Goal-Oriented Action Planning** - Agents with goal-oriented action planning design are working to satisfy goals. Agents perform actions that complete the goal through planning a solution to the goal. Two agents with the same AI logic diverges in behavior based on their environmental perspective and location. Done by creating a plan composed of a list of small, simple actions that when executed sequentially results in the satisfaction of the provided goal [4].

## Tools

**Unity3D** – Unity3D [13] is a game engine designed to quicken and improve the game development process. With built-in libraries for performing complex physics and rendering virtual environments, Unity3d greatly increased productivity. The decision to use the Unity3D Game Engine as the framework for WarGames came as a recommendation. After spending some time working with Unity3D’s features, I found it was more than sufficient for quickly creating a working environment for agents to navigate.

**C#** - Unity3D allows the use of two languages C# and UnityScript. UnityScript, also known as JavaScript for Unity, is a language derived from JavaScript and is currently less popular in Unity3D’s community. C# however, is used by 80.4% of Unity3D’s community because of this, a vast majority of community tutorials and open source solutions are only available in C#. While researching Unity3D, I found this to be the case and settled on using C# as my development language.

**Tactical Shooter AI** – Tactical Shooter AI [12], previously known as Paragon Shooter AI, is the codebase WarGames was designed and built as to extend Tactical. I purchased Tactical Shooter AI from the Unity Asset Store. Then contacted the original developer to receive explicit permission to construct WarGames off his source code, and host it publicly on GitHub. I received the creator’s permission on January 18th, 2016. I chose to build off another person’s work because of time constraints. Tactical Shooter AI provides all basic game logic I needed to begin development of WarGames. Examples include agent sight, movement, and the ability to aim and shoot a gun.

**Visual Studio 2015 Community Edition** – Unity3D provides users the ability to choose from two integrated development environments. Mono’s Monodevelop a very simple lightweight IDE or Microsoft’s Visual Studio [6]. I have chosen to use Visual Studio because it contains a much larger community and features. The community edition of Visual Studio is freely available with enough features to aid in the quick development of WarGames.

# BACKGROUND

Primary research was done into goal-driven autonomous agents. Goal-driven autonomous agent behavior has been well researched and applied in many situations. Primary inspiration is a result of Pattie Maes' *The Dynamics of Action Selection* []. Maes' models an autonomous agent as a set of modules. Successful behavior is defined as ones that are goal-oriented, opportunistic, makes some prediction, and adaptive to changing situations. The overarching goal of her research was to find correct action selection as an emergent property of the module organization.

Other works including goal-driven agents included *Applying Goal-Driven autonomy to StarCraft* [], *Three States and a Plan: The A.I. or F.E.A.R* [], and *Applying Goal Oriented Action Planning to Games* []. All three papers design and implement a goal-driven autonomy solution to a pre-existing game or one they had been developing. Each paper has different methods of creating, storing, and navigating a set of goals and actions. Other differences included the environmental scenarios. Every environment resulted in different sets of actions and variables. Variations of the design architecture were the inspiration to extract the goal creation from the planning agents.

Once I had settled on goal-oriented action planning the works of Héctor Muñoz-Avila, David W Aha, Ulit Jaidee, Matthew Klenk, and Matthew Molineaux *Applying Goal Driven Autonomy to a Team Shooter Game* [] became more relevant. Their work included not only work in goal-oriented action planning, but in the context of teams, and shooter style games.

Edmund Long detailed the increase of agent performance with the introduction of goal-oriented action planning in *Enhanced NPC behaviour using goal oriented action planning* []. Edmund implemented a finite state machine and goal-oriented action planning solution to a scenario and compared their performance attempting to show that goal-oriented action planning was an easy way to improve agent behavior.

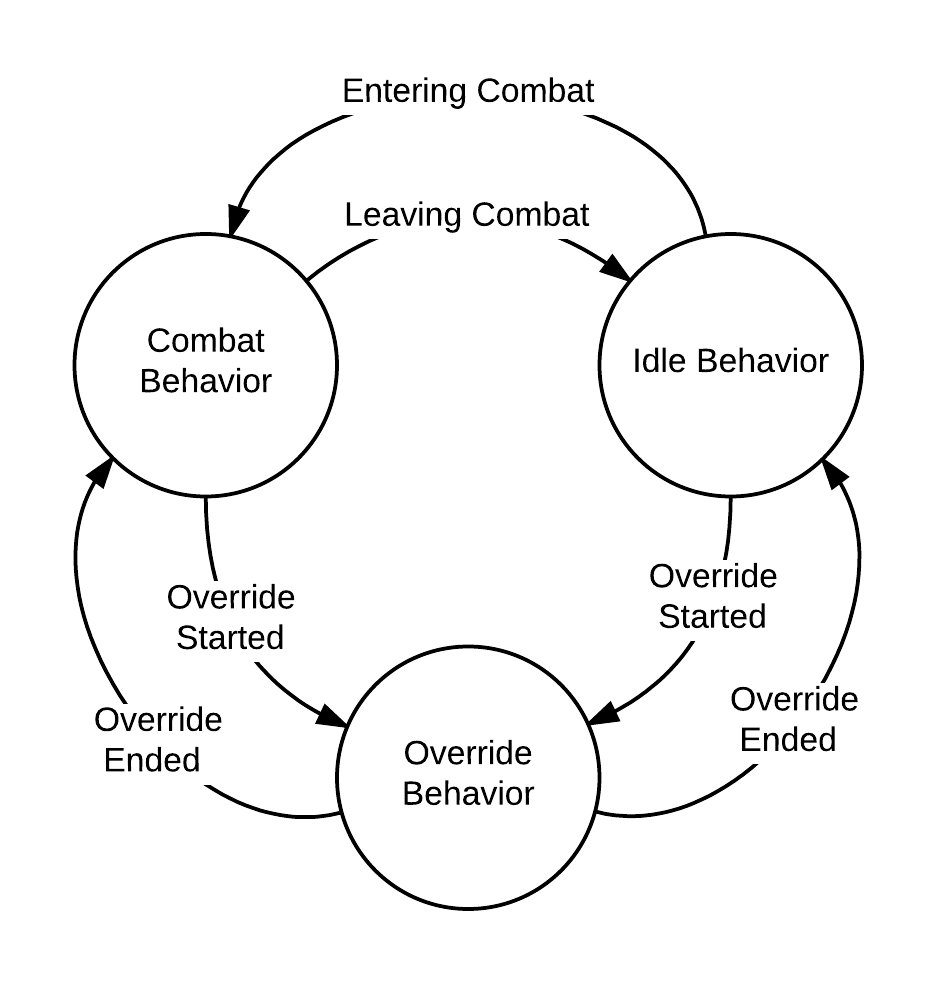
Zhang, Huiliang, and Shen, Zhiqi, and Miao, Chunyan provided a unique take on goal-oriented action planning in *Enabling goal oriented action planning with goal net* []. They focused on the act of goal selection instead of action selection. By organizing goals in a hierarchy based on several goal characteristics they show that typical goal-oriented design is improvable with proper goal prioritization.

In pursuit of previous works on general artificial intelligence information, I drew from various. Stan Franklin and Art Graesser wrote the piece *Is It an Agent, or Just a Program?* [], which contained an overview of their research in how to define the autonomous agent. Gerhard Weiss provided me with my definition of a multi-agent system []. General agent-oriented programming information is credited to Yoav Shoham's [] work in agent systems. Included his work on communication frameworks for multi-agent systems.

Further research into multi-agent systems leads me to Barbara Grosz and Sarit Kraus's [] work on Collaborative plans for group actions.

# DESIGN

## Tactical Shooter AI

WarGames design was created to work with a codebase Tactical Shooter AI (Previously Paragon Shooter AI). Tactical contains all the necessary code needed to create a basic shooter game relatively quickly. One clear advantage of using Tactical as WarGames’ base system was that it is open source and provides a simple interface for including custom artificial intelligence over top the existing architecture.

Tactical runs on a system of behaviors. The two primary behaviors include the idle and combat behaviors, and the third set of behaviors, known as override behaviors, are also included.

**Combat Behavior** – AI logic that runs when an agent is engaging in combat.

**Idle Behavior** – AI logic that runs when an agent is not engaging in combat.

**Override Behaviors** – Short lived segments of AI logic used to perform small tasks. Examples of Override Behaviors include dodging grenades and strafing to avoid gunfire.

A major requirement for the codebase I wanted to use was a large amount of flexibility. Tactical provides me with this flexibility by loosely defining how agent AI can work. Providing a working combat behavior and idle behavior that invoke the extra operations and components necessary for the implementation of WarGames is an easy way to start with a working game logic. Tactical allows extra time for an immediate start of the design and implementation of agent AI.

### Why Tactical does not Solve WarGames’ Goal

In modern combat, much structure exists. Navigation of the battlefield requires significant collaborations between every team member, a command hierarchy determining what soldier control others, and a way for information to flow between each agent.

WarGames establishes team based command hierarchy allowing commanding soldiers to give orders sub-soldiers, a communication network allowing for the passaging of information and orders between soldiers. The addition of these features to Tactical satisfies the requirements of WarGames goal.

## Environment

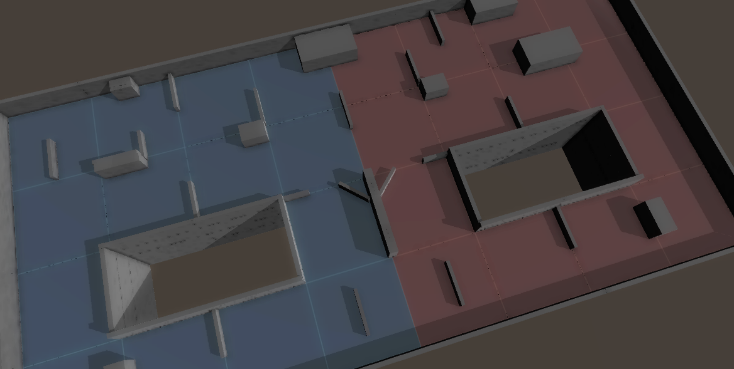


Figure : WarGames Test Environment

The environment that the WarGames’ agents use is a small enclosed battlefield containing several low and high walls used for cover. The battlefield is normally divided equally across the middle to provide each team with an equal chance. Figure 2 shows the environment I used throughout development.

For a team to claim victory in WarGames, one team must accomplish one of two goals. Defeat every agent on the opposing team, or get within a few meters of the opposing team’s wall.

## Goal-Oriented Action Planning

Goal-oriented action planning (GOAP) as described in section 1.1 is a system of artificial intelligence that revolves around the distribution and completion of goals. GOAP allows agents to use information perceived from the agent’s environment to influence how an agent satisfies the requirements defined by a goal. Allowing agents to review every possible method for satisfying a goal based on perspective allows agents to find and use the best method on a case by case basis. GOAP is primarily used in video games to create very dynamic agents capable of performing in a variety of environments [4].

Figure : Tactical Shooter AI Behavior System

### GOAP Key Components

Goal-oriented action planning (GOAP) contains a lot of key components. Terminology for these components can be incredibly confusing contextually. Contained in this section is a general definition followed of any heavily used terminology.

**Goal** – Contains information about the environmental state that the receiving agent should work towards replicating.

**Action** – An action is a small task a soldier can perform. Examples include walking to an area or finding cover.

**Action Plan** – A sequence of actions that when performed sequentially results in the satisfaction of the goal used to create it.

**Planner** – Agents that use GOAP architecture each contains a planner. When provided a goal the planner produces the most effective action plan to satisfy provided goal.

### Actions

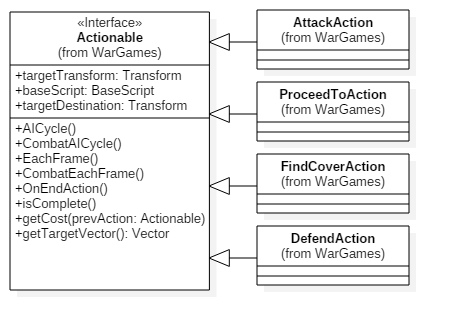


Figure : Actions

WarGames’ actions defined by objects extending the Actionable class which is composed of a single location, combat AI logic, and noncombat AI logic. Combat and noncombat AI logic are the logic ran to perform the task used to whether or not the soldier is engaging in combat with an enemy agent. Lastly, actions return a Boolean value. When the Boolean value is true, the caller knows that the action completed in that usage of the action.

**Attack** – Attack actions are used to satisfy an attack goal. Attack actions prevent the finding of any cover unless an enemy is found, as a result of this attack actions should only be used over a short distance. The location of an attack action defines the area an agent should rush too. Returns true when the agent reaches the action location.

**Defend** – Defend actions are used to satisfy a defend goal. In this action, agents find cover from, but overlooking the defend action location. An agent should have in the view from his cover location. Agents use the first cover they find that has sight on the location. Defend goals imply the anticipation of an enemy thus attempting to defend a location too far away may have negative consequences. Returns true when the agent is in cover overlooking the actions location.

**Find Cover** – Agents using a find cover action attempts to find cover from the actions provided a location. This action contains additional fields. “waitTime” defines how long an agent should use this cover for. “useFirstCover” enables find cover to provide the requesting agent the first correct cover location it finds, thus skipping much time-consuming calculation. This optimization allows the agent to find cover quickly and look for better cover once in a safe location. Agents always find cover with progression to the goal destination in mind. Returns true when the soldier enters cover with sight on the action location.

**Proceed To** – Proceed to actions direct an agent to move in the direction of the actions location. Returns true once the action location is the same as the executing soldiers.

**Wait** – Wait actions do not utilize the location field and require an additional field of “waitTime.” Agents are executing a wait action stay in their current location for an amount of seconds defined by “waitTime”. Returns true when the number of seconds waited exceeds “waitTime”.

### Goals

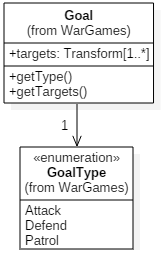


Figure : Goal

Goals in WarGames are composed of a type and a set of locations. Locations represented by a Vector3 object. Goal types defined by an enumerable type variable. This enumerable type contains three possible values: attack, defend, and patrol.

**Attack** – Attack goals provide one location in the set of locations defining where the soldier should navigate too, and when near the goal location agents should move aggressively on it. Useful for when a team leader wants a soldier to get quickly to an area, most likely to aid another agent.

**Defend** – Defend goals also provide on location in the set of locations defining the area a soldier should watch from cover. Soldiers should attempt to reach a cover location overlooking the defend location and wait till an enemy soldier appears the soldier receives a new goal. This goal is useful for team leaders that predict the arrival of an enemy in a location.

**Patrol** – Patrol goals contain two or more locations in the set. Agents should cycle through the locations, arriving on one before moving to the next. Team leaders wishing to gain more information about the environment may set soldiers on a patrol path to gain information on an unknown area.

### Action Plans

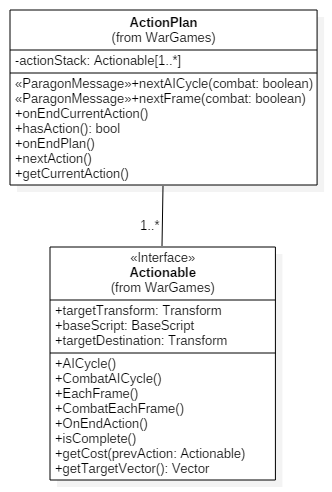


Figure : ActionPlan

WarGames’ action plans comprised of the goal used to create it, an array of actionable objects, and an integer number holding the index to the current action the agent is running. Agents request the next cycle of AI logic to run, and the action plan finds and executes the logic of the current action in the plan based on if the agent is currently in combat. After the execution of an action returns true, the current pointer is increased by one. If the array of actions does not contain an object at the current pointers index, then the plan is complete.

## Agents

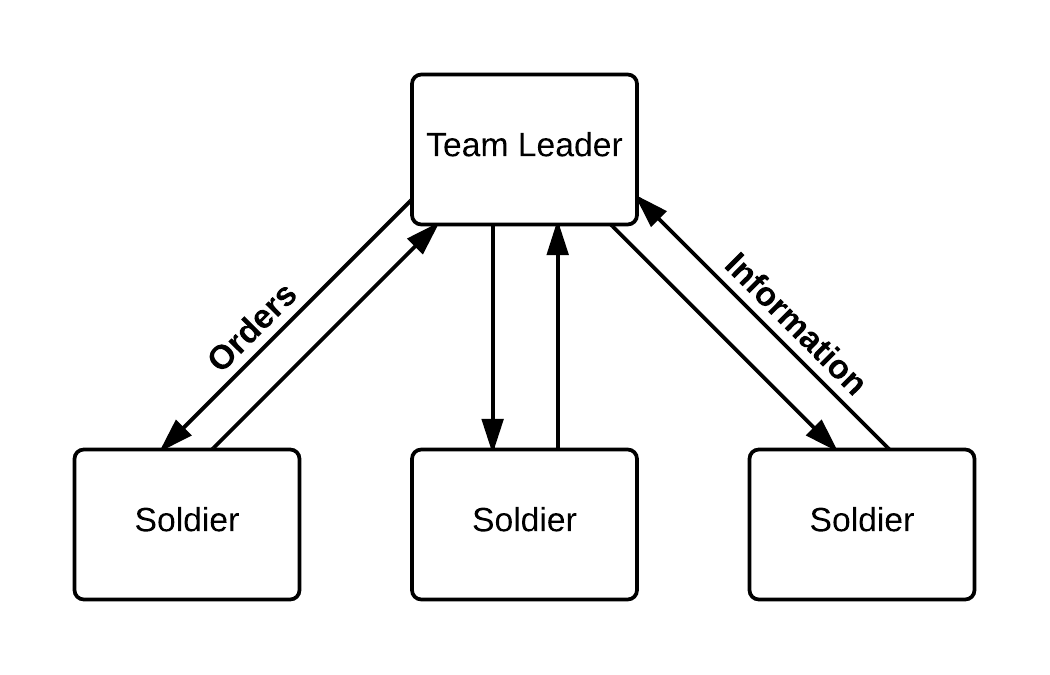


Figure : Team Infrastructure

Agents in WarGames come in the form of two soldier types and team leaders. Each team contains three soldiers and a single commanding team leader. Figure 6 details the command hierarchy of WarGames. Team leaders are allowed to send orders down to soldiers agents. Soldiers can only send battlefield information back to the team leader. Battlefield information for soldiers is perspective based. Each soldier reports his location, damage, and any enemies he currently sees.

I chose this team structure specifically to replicate the US Army. US Army close combat fire teams typically consist of a team leader, rifleman, automatic rifleman, and grenadier. Due to time constraints, I chose to reduce the diversity of team members to a team leader and three generic soldiers.

### Soldier

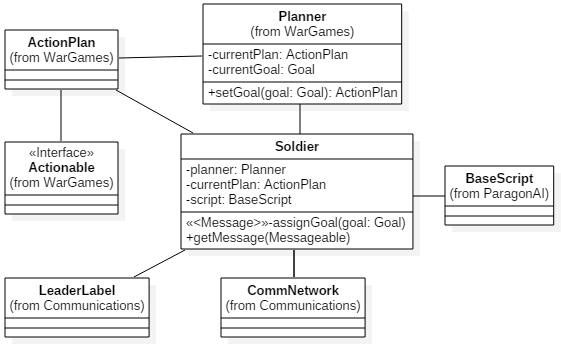


Figure : Soldier Components

A soldier is the most basic agent in WarGames. They receive goals from their team leader and work to satisfy them. Soldiers satisfy goals using Goal-Oriented Action Planning. All components of the soldier include:

**Leader Label** – The leader label is an identification marker on the soldier. This identification can be used to find the soldier’s team and team leader.

**Communication Network** – The communication network facilitates the passing of information from soldiers to team leaders while allowing team leaders to distribute orders to soldiers. The communication described in more detail in section. 3.6.

**Planner** – A GOAP planner capable of generating action plans from goals. In addition to the traditional planner, soldier planners include storage of the soldier’s current goal and plan.

**BaseScript** – The BaseScript is a Tactical Shooter AI script. It contains all methods needed to control the soldier.

### Team Leader

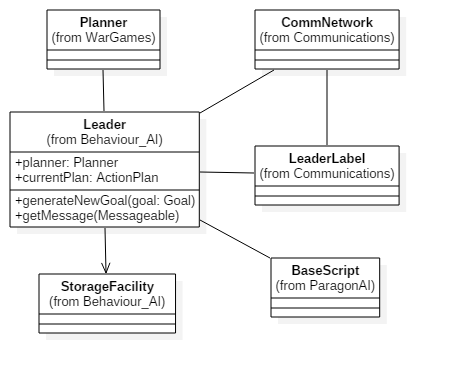


Figure : Team Leader Components

Team leaders control a group of three soldiers. Team leaders perform as soldiers attempting to satisfy goals created by and given to itself. They contain the additional responsibility of created goals for soldiers and keeping track of the environmental state. A team never contains more than one team leader.

Team leaders act as soldiers. In that the same agent that holds the team leader script also holds a soldier script. The team leader creates goals and distributes the goal to itself while also planning action plans and executing them to satisfy this goal. Although this seems ridiculous, it saves me much time by not having to generalize and extract most of the soldier’s logic.

For reasons explained in section 4.3 the team leader’s action plan, actions, goals, and planners were never fully designed. The details provided are slim intentionally. Team leaders contain the following components:

**Leader Label** – The leader label retains the same properties as soldiers along with additional features. The team leader may use their leader label to claim leadership of a team gaining access to distribute orders through the CM.

**Communication Network** – A second communication network to handle the needs of the team leader unlocked with his leader label.

**Storage Facility** – Storage facility is team leader’s component for the reception, storage, and retrieval of information. Information received through the communication network. The storage facility contains a hash table and a linked list.

The hash table contains the information about soldiers from the last information message the team leader received from that soldier. A soldier sends his current health, plan, location, and a timestamp indicating when the soldier gathered the data.

Secondly, the linked list contains reports of enemy locations. Each node in the list contains a location an enemy seen, and the time it happened. A linked list was to allow the team leader to look back as far as he wanted and make assumptions of the current enemy locations.

**Goal** – Team leaders were meant to have a set of goals it would be working to achieve. Team leader Goals would describe a strategy for winning the game. Examples of goals were the rush enemy or play safe.

**Actions** – Team leader actions included giving goals to soldiers and calculating the results of agents executing those goals. Allowing team leaders to estimate what goals would be best.

**Action Plan** – The team leader’s action plan would consist of a growing list of team leader actions. The current action would be goals the team leader is waiting for the completion of soldier’s goals, and actions following that would be the next best goal for that soldier. Provided the team leader has accurately calculated the battle state at the end of the current action for that soldier.

**Planner** – Another GOAP Planner that would take the goal and feed best team leader action plans.

## Agent Communication

Communication has to reflect the hierarchy shown in Figure 6. Team leaders must be able to send orders down to soldiers. Likewise, the soldier must able to send information to their team leaders.

### E:\WarGamesTechnicalPaper\Figures\GiveGoalToSoldier.pngE:\WarGamesTechnicalPaper\Figures\PassInfoToLeader.pngMessageable Classes

Figure : Team Leader Giving a Soldier a Goal

Figure : Soldier Sending Information to the Team Leader

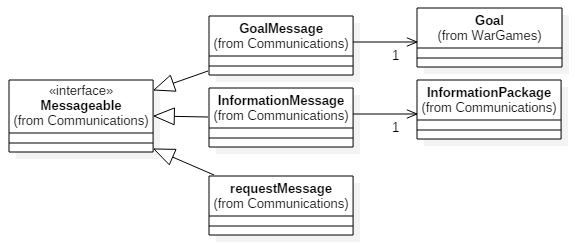


Figure : Messageable Interface and Classes

There are two types of messages a team leader would want to send, one to change the soldier’s goal and another to request information. A soldier should only need to send one type of message containing battlefield information. Thus, WarGames describes a messageable interface which three message classes extend.

**Goal** – The goal message contains a goal the receiving soldier must set as his current goal. The soldier also disposes of his previous goal and action plan.

**Request** – Request messages inform a soldier that the team leader has not received any information from the soldier in a predetermined amount of time, and triggers the soldier to send an information message.

**Information** – Information messages contain information about the agent's current status, and an array of enemies currently in the sight of the soldier, be there any.

### E:\WarGamesTechnicalPaper\Figures\RequestUpdateFromSoldier.pngCommunication Network

Figure : Team Leader Requesting an Update from a Soldier

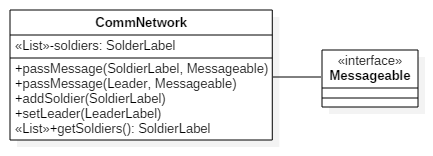


Figure : Communication Network

One communication network shared among all members of a single team. Agents in another team have their respective communication network. The communication designed as a message based system. Where agents leave messages addressed to a team member, and the communication network places it in storage until the agent, it is addressed to requests the next message.

The communication network is composed of:

**Soldier Queues** – Each agent receives a queue that stores messages. This component is a hash table containing soldier objects as keys and their message queue as a value.

**Team Leader Queue** – This is an additional queue to store messages addressed for the team leader.

**Public Interface** – Allows soldiers and team leaders to provide an address agent and message-able object. Then communication network finds and inserts the message into the proper queue on the communications networks own thread.

# IMPLEMENTATION

Due to time constraints, several components of WarGames had to be quickly redesigned for a complete working system to be operational in time. Primarily the soldier agent was almost completely reworked to remove all planning operations and instead run on a simple finite state machine. Other reworked components included team leaders, goals, and dynamic cover discovery.

## Goal

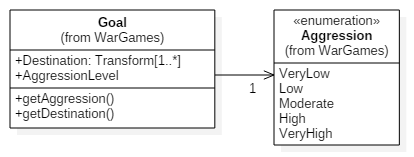


Figure : Goal Implementation

Goals are simplified and now only contain a destination and aggression level.

### Destination

A Vector3 representing a location in three-dimensional space that a receiving agent should make progress. Once at the location agents finds cover within sight of the destination making sure they are taking cover in locations from the direction they approached from facing the enemies starting side of the map.

### Aggression

Aggression is an enumerated type containing five possible values being VeryLow, Low, Moderate, High, and VeryHigh. Aggression affects how an agent behaves when navigating to the destination.

**Very Low** – The agent finds and uses cover at every opportunity, and use that cover for five to nine seconds.

**Low** – The agent finds and uses cover at every opportunity until the agent is fifty percent or more to the destination than agents finds cover at every opportunity. Each cover location used for four to eight seconds.

**Moderate** – The agent does not find cover till fifty percent to the goal location, then until seventy-five percent to the goal location the agent skips every other cover opportunity. Lastly, over the remaining twenty, five percent agents uses cover at every opportunity. The agent uses each cover location for three to seven seconds.

**High** – The agent does not find cover until seventy-five percent to their destination and uses every other cover opportunity from then on. These agents stay in cover for two to six seconds.

**Very High** – The agent does not attempt to find cover. If engaged in combat cover locations are used for one to five seconds.

## Soldier

The soldier agent changed fundamentally from the design. The current set of actions could only produce a limited amount of outcomes. Due to time constraints, I did not have the time to design and implement more actions. To allow me more time to work on other aspects of the systems a simple finite state machine design for the soldier is designed and implemented.

### Actions and Action Plans

Because of the fundamental changes to WarGames in implementation, actions and action plans became obsolete. During implementation I only had designed for the attack, defend, proceed to, find cover, and wait for actions. Also, I had three goals: attack, defend, and patrol.

The low amount of actions and goals created a lack of variety in plans. Almost every plan broke down to finding cover along a path till he reached the goal and attack or defended the location.

Due to the major changes to the soldier’s artificial intelligence model, the soldiers now use goal-oriented AI instead of goal-oriented action planning. The change extracts planning to the team leader and omits actions from WarGames’ design. Soldiers still work towards goal however instead of planning an optimized list of actions they work towards goals using state logic described in the next section.

### Finite State Machine Implementation

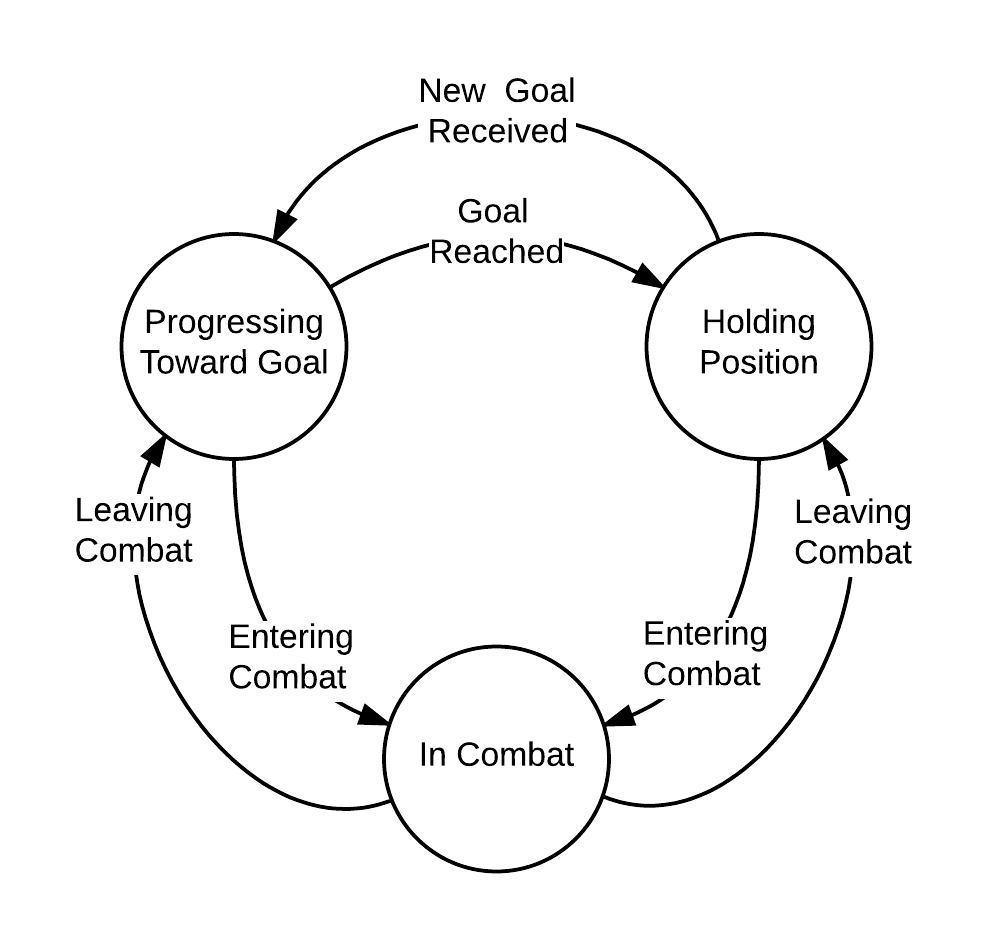


Figure : Soldier Finite State Machine

The soldier agent’s logic now only consists of three states: progressing to a goal, holding a position, and in combat.

**Progressing to Goal** – When the agent is not in combat, and not yet at the goal destination. While progressing to goal agents progresses towards the goal destination finding cover on the aggression level defined in their goal.

**Holding Position** – When the agent is not in combat and has reached the goal destination. Agents holding a position find combat on their team’s side of the map overlooking the goal destination. They continue to use this cover till an enemy agent encountered, or they receive a new goal.

**In Combat** – When the agent is in combat. While engaging in combat, agents find cover in respect to the enemy agent with whom they are engaging but use cover locations that also place them closer to the goal destination.

Soldiers are still autonomous because although receiving direct orders on where and how to move about the battlefield, soldiers still discover proper navigation to a destination from their perspective. Soldiers with the same goal have divergent behavior based on enemy agents in their sight. Behavior also diverges with the soldier’s selection of the amount of time to stay in cover.

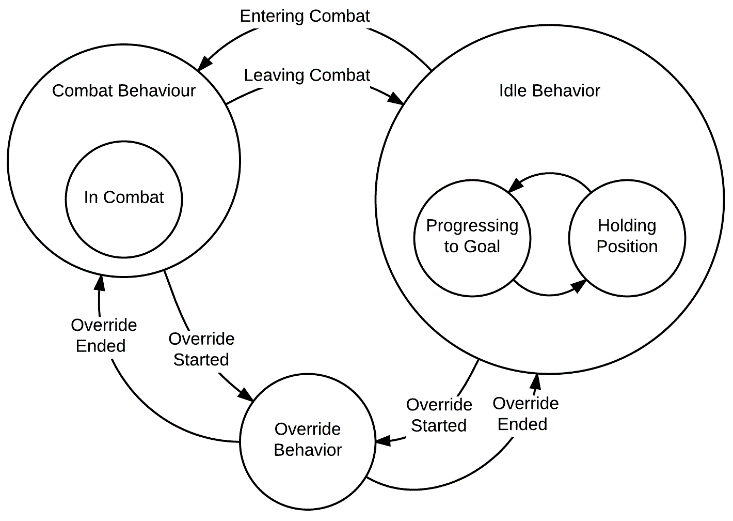


Figure : Adaptation of Tactical's Behavior System to use Soldier FSM

Figure 16 shows the adaptation of Tactical’s behavior system to run the soldier’s FSM. Tactical’s design made the implementation fairly straightforward. The in combat state was placed in the combat behavior while progressing to goal and holding position are placed on the idle behavior. Tactical does include the full source code, so it was possible to change completely how behavior system works. Although there are some obvious benefits to implementing the soldier FSM within the behavior system. Aside from the amount of time saved integration into the behavior system allows soldiers to benefit still from override behaviors and the transitions between combat and non-combat states are already handled. Lastly, I wanted WarGames to be built on top of Tactical not changing how Tactical’s core system works, allowing agents from both systems to coexist. I created two behaviors to implement this structure being CombatPlanningBehavor and IdlePlanningbehavior.

## Team Leader

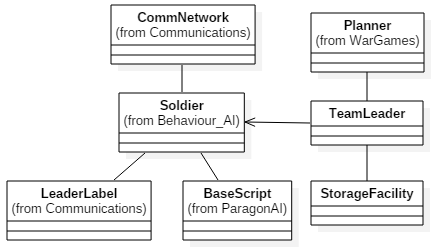


Figure : Components of the Implemented Team Leader

Because the removal of planning by soldiers, any planning that must happen is now executed by the team leader. The team leader looks at environment information currently stored in the storage facility and changes goals for soldiers in reaction to new information. In the implementation, I ran into further time constraints. This lead to me removing team leader actions, action plans, and goals. Simple changes in how a team leader perceived goals, executed planning, and adding a new aggression level component simplified the team leader design. Unfortunately, I have been unable to finish the team leader’s planner. Component changes and additions:

**Goals** – Team leader no longer contains team leader specific goals. Goals, as designed in section 3.3.3, are the new method of ordering soldiers around the battlefield.

**Planner** – The newly theorized team leader planner no longer performs any lookahead. Instead, it provides what soldiers should have what goals. Reacting and recalculating on the discovery of enemy agents, completion of goals, or opportunistic scenarios. The planner sends soldiers to different areas of the map based on the team leader’s aggression level. Opportunistic scenarios would include an allied agent having a clear shot to the victory zone, or an enemy agent extending past an allied agent the enemy has yet to see.

**Aggression** – Although soldiers only have aggression defined in the goal, team leaders contain a personal aggression level. The team leader’s aggression scales how aggressive he makes his soldiers. Higher aggression has team leaders value riskier scenarios as sufficient to qualify as opportunistic.

### Storage Facility

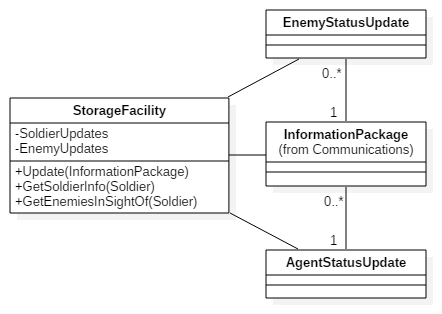


Figure : Storage Facility Components

The changes made to the team leaders storage facility does not change a lot fundamentally about the kind of information is stored, but does change how some data is accessed. This implementation was also quicker and provided the team leader the ability to retrieve clustered groups of data about each agent on the battlefield.

**Soldier Status** – Only one change happened to the storage of received soldier information besides the development of storage facilities design. The hash table takes a soldier as a key with an “AgentStatusUpdate” object. The status update contains the soldier’s health, current goal, percent to goal, and a time stamp.

**Enemy Status** – In the implementation I changed the linked list implementation of enemy locations to the secondary hash table. This hash table also takes a soldier object as a key, and an array of “EnemyStatusUpdate” objects. When a soldier sends an update to the team leader, he sends an enemy status update for every enemy agent currently in the sight of the sending soldier.

## Communication Network

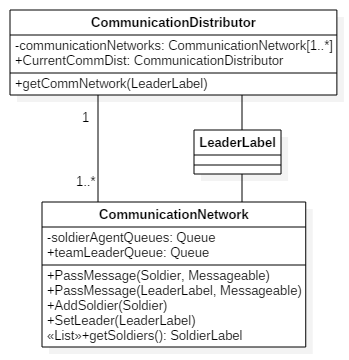


Figure : Communication Distributor & Network Relations

The only changes from the design that communication networks received was their method of distribution. All members of the same team must contain a reference to the same communication network object. Facilitating of the correct distribution of communication networks lead to communication distributor’s design and implementation. Only one distributor can exist at any one time. Upon the creation of an agent they grab the reference to the distributor and request the reference to the communication network for their team. Agents provide their leader label to the distributor, and the distributor provides the reference to the communication network paired with the provided leader label.

## Dynamic Cover Discovery

Originally all WarGames agents were designed to use Tactical’s methods for dynamically finding cover locations. When the implementation of WarGames began it became obvious that Tactical’s provided methods of finding cover dynamically wouldn’t work for a variety of reasons. Tactical’s dynamic cover discovery was built only to find cover from one specific location. This cover selection is bad because WarGames’ agents need to switch progressively cover towards a destination. Tactical’s scripts also didn’t account for the direction an agent is traveling so agents attempting to hold a position would take cover on the wrong side of the map letting enemy agents dispose of them instantly. To solve these problems a new find cover script “WarGamesFindCoverScript” was implemented. The logic in this script was derived from Tactical’s “FindCoverScript” but includes significant changes. Addition features in WarGames’ find cover script include taking in account the direction of agent travel, cover selection based on aggression, progressive cover selection.

### Navigation Mesh

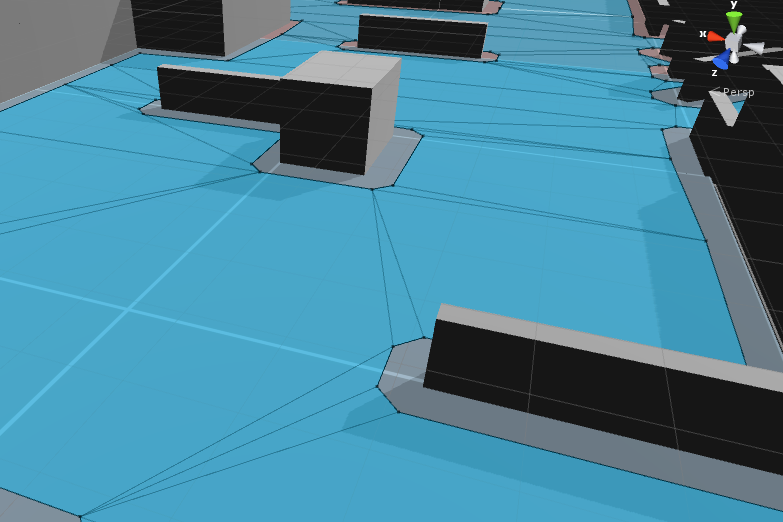


Figure : Sample Environment with Navigation Mesh Overlay

In Unity3D agents can move about an area using what is called a navigation mesh. Figure 20 shows the navigation mesh overlaid on a sample environment. The navigation mesh contains three components for defining the area it occupies. Vertices which are single points on the battlefield represented as black dots. Edges which connect two vertices, shown as black lines. Lastly polygons blue areas occupying the spaces between edges.

Notice that every vertex is near a section of wall. This placement makes them perfect for calculating possible cover locations. Provided no enemy agents are near allied soldier the allied soldier could move from vertex to vertex toward his goal destination. Upon discovering an enemy agent, the soldier may already be in cover. If the agent is not in cover, then a quick transition to a nearby vertex should provide necessary cover.

### New Dynamic Cover Discover Algorithm

WarGames’ find cover script requires some additional parameters to be able to calculate a proper cover position.

**Goal** – Used to grab aggression level and destination.

**Holding Position** – Boolean value to dictate how the script should treat the goal destination. True indicates that the agent needs to find cover overlooking the goal destination. False indicates the agent should find cover towards the goal destination.

**Direction** – Character array of size two. Used to calculate what side of the battlefield the agent is approaching. Thus removing the possibility of an agent mistakenly taking cover on the wrong side of the map. Index zero contains x or z. The character x indicates the agent is moving along the x-axis. Respectfully z shows the agent is moving along the z-axis. Index one contains l or h. An l character, short for Lower, represents that any cover found should have an x (or z respective of index zero) should be a lower value than the goal destination. H, short for higher, means any cover should have a higher value than the goal destination.

**Percent to Goal** – Necessary because aggression defines how an agent should find cover based on how close the agent is to the goal. As defined in section 4.1.2

The WarGames find cover script follows the logic

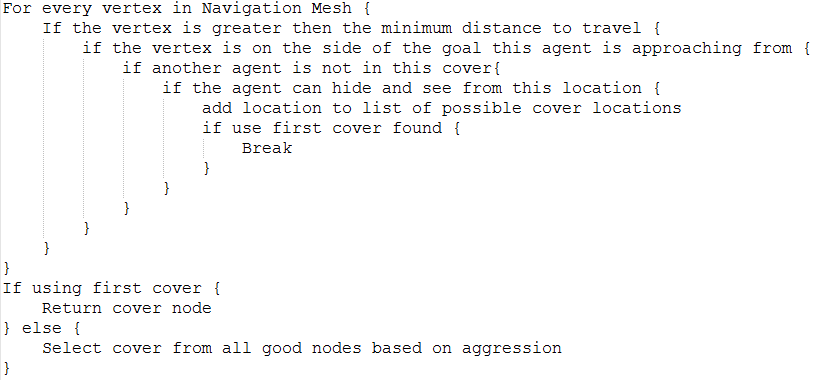


Figure WarGames Find Cover Script Pseudocode

# EVALUATION & CONCLUSION

## Issues that Became Important

During implementation, I was faced with two major issues. Time constraints and dynamic cover discovery. The details below reflect why WarGames implementation resulted as described in section 4.

### Time Constraints

After I finished the design of WarGames and began to implement its components I noticed the remaining time to complete WarGames was fleeting. After only a short amount of implementation I had a discussion with my professor, Dr. Joshua Gross, which brought me to the realization that the designed solution was more complex than it needed to be. Provided I had more time to work on the project I could have created far more actions making a GOAP implementation worthwhile. However, due to the ever increasing deadline I needed to simplify the soldier agent. The soldier’s FSM implementation provides strikingly similar behavior for significantly less implementation time. The development of the soldier’s FSM resulted in all planning being removed from soldiers and supplemented with additional planning from the team leader.

Team leaders changed slightly in how planning would happen. Due to soldiers no longer doing any long term planning team leaders would have to provide soldiers short term goals. As a result team leaders needed to keep a closer watch on soldier progression and enemy discovery. Quick response to changes in the soldier’s state and environmental perspective would provide a more lifelike intelligence model. For example, an allied soldier who sees an enemy soldier would have a higher chance of a successful engagement if the team leader sent another nearby soldier to his aid.

### Dynamic Cover Discovery

Dynamic cover discovery quickly became an issue when I noticed Tactical’s implementation would not work. The script required a rewrite that I was completely unprepared to do. I had to identify quickly why Tactical’s cover script did not work and design a solution I could complete while staying within time constraints. Rewriting this script was a difficult problem because I had to make time for an issue I budgeted no time for, while already behind where I wanted to be. I strongly believe if Tactical did not contain a basic dynamic cover discovery method I would not have completed the amount of work discussed.

## Design Elements that didn’t fit

As discussed throughout the paper my primary limiter was time. As a result, soldiers had actiontionable classes, action plans, and planning components removed from their design. Resulting in an entirely new AI model for the soldier to use.

Team leaders also saw significant changes. As discussed in section 3.4.2 team leaders were to have their goals, actions, and plans specifically for its use. Further design resulted in an AI model closer to that of a reactive agent with very limited planning abilities.

## Statement for Future Work

### Team Leader

The planning component of the team leader has never seen implemented. For WarGames to be considered complete, the team leader needs at minimum basic planning functionality to be able to create and distribute goals for soldiers.

In addition to the planning module, the ability for soldiers to be able to take command of a team when the current team leader dies remains a problem. Several methods of implementation discussed never made it to paper due to the low priority over other aspects of WarGames.

### Submission for Review

After completing the work on team leader and performing quite a bit of code clean up, I plan on submitting WarGames to the original creator of Tactical. Throughout the design and implementation of WarGames, I have been very careful to do minimal editing to any of Tactical’s code. Integrating my work into Tactical’s asset packaging should prove straightforward. Allowing me to see the fruits of my labor in use, and provide other customers of Tactical a significant feature increase.

### Game

The nature of my project points directly towards the creation of a game. The user would be the team leader allowing them to create and distribute their goals among soldiers of their team. Facing the user against a WarGames team leader in a variety of situations.

# ACKNOWLEDGMENTS

Dr. Joshua Gross for helping me significantly with expertise in artificial intelligence, teaching me proper UML, always making himself extremely available to give me assistance, and lastly for never forgetting to tell me how pleased he was with my progress after meets. Especially meetings that included huge refinement of my work, which without reassurance would have surely left me very disappointed.

Dr. Kevin Coogan for an incredible amount of encouragement throughout my entire project and providing me with the necessary practice of properly describing the components of my project. A skill that became very useful when preparing my presentation, and will remain useful for out my entire career.

Additional acknowledgments to Dr. Joshua Gross and Dr. Kevin Coogan for insisting I take steps to improve my writing skills. I have learned more grammar in the past year than twelve years of public education.

Jack S for allowing me to use and publicly host the source code of Tactical Shooter AI for my project. Tactical significantly improved the overall quality of my project. I can only imagine the horror of a project I would have produced in attempting to build an entire game from scratch.

My peers Nathan Wilson, Elizabeth Quick, and Mathew Pfannenstiel for providing support and critiques throughout the entirety of our seminar experience.

Shelby Rainford for her amazing copy editing skills and support throughout development. Consistently reminding me her help was available and putting up with tight deadlines out of pure kindness. I just wish I had time for her to review my final submission. Any grammatical errors are no fault of hers.

My best friends Andrew Kirby, Bryan Bethel, Caleb Long, and Josh Thompson for continuous support and forcing me to get enough sleep.

Katrina Jahn for additional copy editing of my submissions.

# REFERENCES

[1] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In *Intelligent agents III agent theories, architectures, and languages*, pages 21–35. Springer, 1997.

[2] Barbara J Grosz and Sarit Kraus. Collaborative plans for complex group action. *Artificial Intelligence*, 86(2):269–357, 1996.

[3] Bohemia Interactive. *Virtual Battle Space 3*, 2016.

[4] Edmund Long. *Enhanced NPC behaviour using goal oriented action planning*. PhD thesis, University of Abertay Dundee, 2007.

[5] Pattie Maes. *The dynamics of action selection*. Artificial Intelligence Laboratory, Vrije Universiteit Brussel, 1989.

[6] Microsoft. *Visual Studio*, 2015. Available at https://www.visualstudio.com.

[7] Héctor Muñoz-Avila, David W Aha, Ulit Jaidee, Matthew Klenk, and Matthew Molineaux. Applying goal driven autonomy to a team shooter game. In *FLAIRS Conference*, 2010.

[8] Department of the Army. *The Infantry Battaloin FM 3-21.2*, 2006.

[9] Jeff Orkin. Applying goal-oriented action planning to games. *Monolith Productions*, pages 1–11, 2003.

[10] Jeff Orkin. Three states and a plan: the ai of fear. In *Game Developers Conference*, volume 2006, page 4, 2006.

[11] Yoav Shoham. Agent-oriented programming. *Artificial intelligence*, 60(1):51–92, 1993.

[12] Squared55. *Tactical Shooter AI*, 2015. Available at https://www.assetstore.unity3d.com/en/#!/content/47248.

[13] Unity Technologies. *Unity3D Power*, 2015. Available at http://www.unity3d.com/Pages/2d-power/.

[14] Ben George Weber, Michael Mateas, and Arnav Jhala. Applying goal-driven autonomy to starcraft. In *AIIDE*, 2010.

[15] Gerhard Weiss. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT press, 1999.

[16] Huiliang Zhang, Zhiqi Shen, and Chunyan Miao. Enabling goal oriented action planning with goal net. In *Web Intelligence and Intelligent Agent Technologies, 2009. WI-IAT’09. IEEE/WIC/ACM International Joint Conferences on*, volume 2, pages 271–274. IET, 2009