The Behavior Oriented Design of an Unreal Tournament Character

Samuel J. Partington and Joanna J. Bryson

Department of Computer Science, University of Bath, Bath BA2 7AY, United Kingdom
sam@samsolutions.co.uk, J.J.Bryson@bath.ac.uk

Abstract. This paper presents a case study for using a relatively recently developed methodology, Behavior Oriented Design, to develop an Intelligent Virtual Agent (IVA). Our usability study was conducted in Unreal Tournament using the game Capture The Flag. The final agent displays reasonably competent behavior: she is able to pursue multiple goals simultaneously and produce well-ordered behavior.

1 Introduction

This paper presents a case study of the application of a recently-established methodology for developing complex humanoid agents to the problem of building a game agent. The methodology is Behavior Oriented Design [7, 8]. The game is Unreal Tournament [12], using the Gamebots interface [13].

We begin this paper with some background description of both the game and the development methodology. Then we describe the development of the robot, highlighting the elaboration of its action-selection network as the agent becomes more complex. This strategy is taken because it is fairly intuitive, since action selection determines the priorities of an agent. However, Behavior Oriented Design is at least as much about building the behavior objects that actually control the agent's actions, perception and memory as it is about the problem of action selection. The final section goes into detail about the trickier elements of building behavior for this agent, and shows how these problems interact with the problem of action selection.

2 Background

2.1 The Game

This case was conducted using the Capture the Flag game-mode of Unreal Tournament (UT). Unreal Tournament [12] is a First-Person Shooter (FPS) game. As the name suggests, the viewpoint adopted by the player in FPS games is that of the character he or she is controlling: the player sees the world through the character's eyes. The single-player version of Unreal Tournament pits the human player against computer-controlled players ('bots') in kill-or-be-killed deathmatches spread over a wide range of expansive 3D environments. In Capture-the-Flag mode, two teams (or possibly two single players) compete against each other. Each team has a base in which their flag is located.

The object of the game is to obtain your opponents' flag (done by running into it), and return with it to your own flag. This counts as a flag capture. Once a specified number of captures have been achieved, the game is won. If the opposing team captures your flag, you must recover it before you can make a successful capture, as returning to your base with the enemies' flag achieves nothing if your own team's flag is not there. Once a player has captured a flag, s/he may be forced to drop it by being killed (using the usual UT weaponry). The flag then lies on the ground waiting for someone (of either team) to pick it up. If you pick up your own flag dropped by an escaping enemy, it returns to your base instantly. Teams in CTF may be composed of human players alone, or of a mixture of human and computer players.

2.2 The Methodology

Behavior-Oriented Design (BOD) is a methodology for complex agent construction. It derives from the traditions of both Behavior-Based AI [2, 4, 6] and Object-Oriented Design (OOD) [3, 10, 16] the notion of strong modular decomposition. Each module (encoded as a class in an OO language) is semi-autonomous. The purpose of a module is to produce and control expressed behavior, but they also encapsulate whatever memory and perception is necessary for that behavior, and and whatever additional methods are necessary for maintaining the state of the memory or processing the perception and control.

Modular systems require some form of coordination between the modules to guarantee overall coherence for the agent and to arbitrate in cases where behavior modules would express conflicting actions (e.g. those that require going in two directions at once.) BOD uses Parallel-rooted Slip-stack Hierarchical (POSH) dynamic plans[1] encoded in a script file to do this arbitration.

BOD is an iterative development methodology. The iterations begin with an initial decomposition for the agent:

1. Specify at a high level what the agent is intended to do.
2. Describe likely activities in terms of sequences of actions. These sequences are the the basis of the initial dynamic plans.
3. Identify an initial list of sensory and action primitives from the previous list of actions.
4. Identify the state necessary to enable the described primitives and drives. Cluster related state elements and their dependent primitives into specifications for behaviors. This is the basis of the behavior library.
5. Identify and prioritize goals or drives that the agent may need to attend to. This describes the initial roots for the dynamic plan hierarchy (described below).
6. Select a first behavior to implement.

[1] Dynamic plans were historically referred to as 'reactive plans', because they responded rapidly to the environment. Unfortunately, this has lead some people to believe (falsely) that agents that use them are not *pro*-active. Since our agents all have their own goals and motivations, we have adopted this new nomenclature.

Getting the decomposition right the first time is neither critical nor expected — the iterative process will involve refactoring this decomposition. The lists compiled during this process should be kept, since they are an important part of the documentation of the agent.

The heart of the BOD methodology is an iterative development process:

1. Select a part of the specification to implement next.
2. Extend the agent with that implementation:
   - code behaviors and dynamic plans, and
   - test and debug that code.
3. Revise the current specification.

BOD's iterative development cycle can be thought of as sort of a hand-cranked version of the Expectation Maximization (EM) algorithm [11]. The first step is to elaborate the current model, then the second is to revise the model to find the new optimum representation. Of course, regardless of the optimizing process, the agent will continue to grow in complexity. But if that growth is carefully monitored, guided and pruned, then the resulting agent will be more elegant, easier to maintain, and easier to further adapt.

2.3 BOD Action Selection

Dynamic plans support action selection. At any given time step, most agents have a number of actions which could potentially be expressed, at least some of which cannot be expressed simultaneously, for example sitting and walking. In architectures without centralized action selection, such as the Subsumption Architecture [4] or the Agent Network Architecture (ANA) [15], the developer must fully characterize *for each action* how to determine when it should be expressed. This task grows in complexity with the number of new behaviors. For engineers, it is generally easier to describe the desired behavior in terms of sequences of events.

Of course, action-selection sequences can seldom be specified precisely in advance, due to the non-determinism of environments, including the unreliability of the agent's own sensing or actuation. Several types of events may interrupt the completion of an intended action sequence. These events fall into two categories:

1. some combination of alarms, requests or opportunities may make pursuing a different plan more relevant, and
2. some combination of opportunities or difficulties may require the current 'sequence' to be reordered.

Thus the problems of action selection can be broken into three categories: things that need to be checked regularly, things that only need to be checked in a particular context, and things that do not strictly need to be checked at all.

BOD uses dynamic plans to perform action selection through behavior arbitration. BOD dynamic plans provide three types of plan elements corresponding (respectively) to the three categories of action selection mentoned above. A *drive collection* provides the main loop of the action selection, continuously monitoring which drive should be attended to currently. A *competence* checks for context-specific behaviors, and action patterns encode true sequences.

There is a great deal more to be said about POSH action selection — most significantly the importance of prioritizing the elements of a competence in a way such that they converge. Some of this will be elaborated below. There are also many more details of the BOD development methodology, such as heuristics for determining when the complexity of a plan should be offloaded to a behavior, and *vis versa*. These details have been previously published [7–9]. We have also previously published extensive comparisons between BOD and related architectures [7, 9, 17].

The purpose of the present paper is to provide a case study of applying these rules. This serves both to clarify previous publications through an additional example, and also to illustrate the application of BOD to the important real-time domain of computer games. The code for this paper was written using the Kwong [14] python-based implementation of POSH, known as pyPOSH.

3 The Bot in Action

This section presents a number of scenarios demonstrating the actions of the bot we created (the *bodbot*) and relates these back to the plan files created. These scenerios are ordered to show iterations of the development cycle, thus they show bots capable of increasingly complex behaviour. BOD agents are generally referred to by the name of their POSH scripts, because the script determines an individual agent's priorities. Thus quite different agents can use the same BOD behaviour library — indeed, testing old scripts after elaborating the behavior library is part of the BOD iterative development cycle.

The section's purpose is threefold:

- To demonstrate the development of the plan files.
- To illustrate how the actions of the bot are guided by the plan file it uses.
- To give examples of the bodbot's actions, and thus provide a starting-point for the discussion of the development process.

The actions of the bot are illustrated by a series of commentary-style descriptions which are interleaved with brief analysis and samples of plan code. For brevity, only particularly noteworthy parts of the bots' runs are described. The first plan is illustrated by the actions of a male bot on the red team and remainder by the actions of a female bot on the blue team.

3.1 Walking To Navigation Points

We started from a bot based upon *poshbotfollow.lap*, the plan created by Kwong [14] for his "poshbot". *poshbotfollow.lap* had the bot wandering around and following any players he saw. Our initial plan removed the player-following element, replacing it with one which attempted to follow navpoints (navigation-points, aka pathnodes):

> *Yes, the bodbot has just this moment spawned into the play-area. He's wasting no time running off that ledge and towards the tunnel, seems to be having a bit of trouble on the corners, though: he's paying more attention to that wall than*

it really deserves... no, here he goes off again. Looks like he's missed that vital turning though, seems more interested in the walls of the tunnel again, no wait, he's coming back, takes the turning, now he's looking around again, trying to decide where to go. He's finally decided and now he emerges from the tunnel.

The important part of this plan is the competence below. (The top level of the plan hierarchy (the *Drive Collection*) only contains two drives at this point and thus almost always fires this competence as the other is only triggered when the bot walks into something.)

$$\textbf{get-to-enemy-base} \Rightarrow \left\langle \begin{array}{r} \text{(at-enemy-base)} \Rightarrow \textit{goal} \\ \text{(reachable-nav-point)} \Rightarrow \text{walk-to-nav-point} \\ () \Rightarrow \text{wander-around} \end{array} \right\rangle \quad (1)$$

A *competence* is essentially a focused set of productions, each associated with a priority as well as a trigger, and a habituation factor (described later.) The first (highest priority) element of this competence is its goal — triggering it causes the competence to terminate. The second element is intended to find the base, and the third to generate wandering behavior until the second element's trigger can be achieved.

When the bot starts up, he can see a navigation point specified as reachable (the *reachable-nav-point* trigger returns true) and so he runs off the ledge (only a short drop) to get to it. On the occasion of his trouble in the tunnel, the problem is that because of the curve of the tunnel he can no longer see any navpoints. For this reason the lowest-priority element takes over (an empty trigger means that it always fires if no higher-priority element can). This element triggers the *wander-around* competence, which causes the bot to walk around near (and into!) the walls as described. For brevity, this competence is not given here.

3.2 A Greater Awareness of Flags

And here comes the blue bodbot now. She's looking around, wondering where to go next. And now she's off, running towards the tunnel...

The "looking around" at the beginning comes from a modification to the *get-to-enemy-base* competence, whose elements are now the following (the second is new):

$$\textbf{get-to-enemy-base} \Rightarrow \left\langle \begin{array}{r} \text{(at-enemy-base)} \Rightarrow \textit{goal} \\ \text{(reachable-nav-point)} \Rightarrow \text{walk-to-nav-point} \\ () ::10 \Rightarrow \text{rotate} \\ () \Rightarrow \text{wander-around} \end{array} \right\rangle \quad (2)$$

Although two of the elements have triggers that succeed by default, the retries limit (*10*) on *find-nav-point* means that the lowest-priority element does sometimes get a chance to fire. In the example given above, however, the rotating leads to a position where the bot can see a reachable navpoint, and thus the first non-goal element fires.

The bodbot emerges from the tunnel, she's almost at the enemy base now, the prize in her sights. Yes, I think she's going to make it! She makes a clear run for the red flag and grabs it! Nice work there, but can she capitalise on this early success? Remember, she's still got to take it home.

To understand the bot's next actions (running to the enemy flag), we need to consider the, now extended, top-level Drive Collection, **life**:

$$\left\langle\left\langle \begin{array}{r} \text{(our-flag-on-ground)} \Rightarrow \text{go-to-own-flag} \\ \text{(enemy-flag-on-ground)} \Rightarrow \text{go-to-enemy-flag} \\ \text{(see-enemy-with-our-flag)} \Rightarrow \text{attack-enemy-with-flag} \\ \text{(enemy-flag-reachable) (have-en.-flag } \perp\text{))} \Rightarrow \text{go-to-enemy-flag} \\ \text{(hit-object) (rotating } \perp\text{))} \Rightarrow \text{avoid} \\ \text{(have-enemy-flag)} \Rightarrow \text{go-to-own-base} \\ \Rightarrow \text{get-to-enemy-base} \end{array} \right\rangle\right\rangle \quad (3)$$

Life of course has no goal and should in theory never end, but otherwise a drive collection is much like a competence, except that its elements are checked on every iteration of the action selection in case a different drive element should take priority. Before elaborating the drive collection, the main element firing had been *to-enemy-base*. Now, once the bot approaches the enemy flag, however, the trigger *enemy-flag-reachable* returns true and *go-to-enemy-flag* is fired instead.

> *Wait a minute, John, there seems to be some sort of upset at the other end of the arena! Yes, the bodbot's quest for glory has left her own flag dangerously unguarded and the red player has stolen it!*

To demonstrate a situation more similar to genuine Capture the Flag games, I intervened at this point and, playing as the red player, stole the blue flag.

> *The bodbot's leaving the tunnel now Clive, she's surely going to notice that thief any second now...*
> *Too right, John, the bodbot rounds on the red player, running towards him and shooting and ... it's a success! He's been tagged, and he drops the blue flag to the ground where the bodbot grabs it, restoring it to its rightful place! Yes, nothing can stop her now! She's running back to her own flag, she's made it now, the blue team scores!!*

In an attempt to get the blue player to notice me (and since I cannot win whilst the other team has my own flag), I returned to the blue player's base. The bodbot then noticed that I had her team's flag. Doing so meant that her current action of going home was interrupted as the *attack-enemy-with-flag* Drive Collection element fired instead (it has a higher priority) and the bot began to attack me.

Upon being tagged (killed), the red player drops the blue flag he has been carrying and the bodbot's current undertaking is again interrupted, as the *go-to-own-flag* element now fires (it has an even higher priority). Picking up one's own flag returns it instantly to the base, and the bodbot scores when returning to her own flag while carrying the red one. The bot only moves towards her own flag as the list of navpoints leads there: at this stage there is no specific drive to run directly there once it is reachable.

> *Well, that certainly was impressive. The bodbot seems to have had enough though, she's not going anywhere! This is remarkable, she's just standing there! What it she thinking?!*

This final segment illustrates a problem: the expiry of out-dated state the bot holds. In this case, the instance of the *PositionsInfo* class held out-dated information about the enemy flag, claiming that it was reachable from the bot's current location (as that had the been the case until the bot scored and the red flag was returned to the red base). The bot therefore attempted to send a command to make it run directly to the enemy flag. This was not possible from its current location, and so nothing happened.

Out of date state is one of the reasons reactive AI proponents used to avoid all memory whatsoever, but such a strategy is pointless when an agent needs to learn and perform complicated tasks. Under BOD, the correct thing to do is to redesign the system on the next iteration to fix the bug.

3.3 Responding to Attack

This final scenario introduces a number of new elements, the most important being the bot's ability to respond when it is attacked.

> *For those of you who've just joined us, we're seeing a fine run by the blue bodbot, she's just grabbed the red flag! But where are the defence? Well, someone's trying to shoot her but not doing a very good job of it, that shot landed just in front of her. The bodbot's off again now, and ouch! That goo-explosion's got to hurt.*

The assailant was a bot controlled by me. The goowand fires blobs of goo which stick to walls and floors and remain there for a few seconds before exploding.

> *Not one to let that sort of behaviour go unnoticed, she's looking around for the assailant, she's spotted him now and begins to shoot... ooh, right in the stomach! Keen not to throw that lead away though, she's now heading back to her own base. Obviously doesn't want another surprise attack, she's keeping firmly focussed on that attacker as she runs back through the tunnels.*

The response to attack comes as a result of the following new Drive Collection element:

$$\text{(damaged)(armed-\&-ammo)(responding-to-attack } \bot) \Rightarrow \text{respond-to-attack} \qquad (4)$$

This element has a higher priority than go-home, the drive element previously being attended to, and so the *respond-to-attack* competence is triggered. Note again this is substantially different from a normal dynamic plan — the last conjunct should not be necessary, the response to attack should be continuous under philosophies such as subsumption architecture. However, in this 'real world', actions not only have duration, but can only be sent to the game engine once in a while, so the robot has to maintain state to ensures he doesn't flood the game engine.

In some cases, the bot will receive details of the assailant when receiving a message from Gamebots about damage inflicted. For example, if the bot actually sees the shot being fired. This was not the case in this example, however, and so *respond-to-attack* triggers the following competence:

$$\textbf{find-attacker}\ (::\ 3\text{sec}) \ \Rightarrow \left\langle \begin{array}{c} \text{(see-enemy)} \Rightarrow \text{respond-to-visible-attacker} \\ ::1 \Rightarrow \text{big-rotate} \end{array} \right\rangle \qquad (5)$$

This competence is the reason the bot looks around for the attacker: the *big-rotate* element causes the bot to spin. Note the limit on retries here: the bot shouldn't keep on turning around as it may never be able to see the attacker. Further, this competence has no goal, but automatically times out after 3 seconds even if it is responding, to allow some other drive to take over the situation. In this case though, the search was successful, leading to the *see-enemy* sense returning true and the *respond-to-visible-attacker* element running. It is this element which makes the bot shoot the attacker. So the first element in this competence is actually somewhat redundant.

Finding an attacker results in variables being set telling the bot to keep looking at the attacker whilst performing other actions. In practice, this means that when running, the bot instead sends a command to *strafe*. Strafing is running in one direction while facing another.

> *Into the home strait now, she turns around for the final sprint, she's nearly there, yes ... she scores! Now she's going back to try another capture, it could be a high-scoring game, folks!*

The Drive Collection used for this scenario contains three unexciting but nevertheless very important elements: those which expire state. For example, the reason that the bot now goes back for another capture rather than just standing around as before is the following element:

$$freq : 20\text{sec} \Rightarrow \text{expire-the-reachable-info} \tag{6}$$

The expiry elements are the highest priority in the Drive Collection. However, their limits on frequency mean that other elements get plenty of chance to run.

4 The Development Process

The previous section focussed a great deal on action selection. This is a natural consequence of the fact that the dynamic plan scripts essentially determine the goals and motivations for an agent by ordering its priorities. However, there would be nothing to order if it weren't for the behaviour modules which provide the primitive action and maintain the agent's internal state / memory.

4.1 Behaviour Modules

The bot's expressed behaviour is generated by four primary modules, each of which is stored as a separate Python class:

- Movement: state to do with positions of objects, bases and the bot himself.
- Status: contains state regarding health level, weapons held and so on.
- Combat: state about who is attacking the bot, what enemies are around and what teammates are around.
- AndyBehaviour: primitives developed for the 'poshbot'.

Our bot makes much use of code from 'the poshbot', an Unreal Tournament agent designed by Kwong [14] as part of the development of pyPOSH. Although this meant that the behaviour decomposition was not as logical as it could be (many of these primitives would be logically suited to the movement module instead), we felt that such a distinction between the simple behaviour of the original bot and the more advanced behaviour of the bot I developed was useful.

There were also three behaviours dedicated primarily to maintaining internal state. These were made individual behaviours because their state was utilised by more than one of the other behaviours, so could not be seen as an attribute of just one of them.

- Bot_Agent – general information from the Gamebots interface, also inherited from Kwong.
- CombatInfoClass – holds state relating to combat (for example, details of the player holding the bot's flag), and is used by both the Movement and Combat behaviours.
- PositionsInfo – holds state relating to the position of the bot and position of the game objects (e.g. flags and navigation-points), and is used by Movement, Status and Combat.

Like the primitive-complexity *vs.* plan-complexity tradeoff, there is also a trade-off between plan-complexity and the amount of state required. Bryson [7, section 6.5] gives the example of an insect which could either have two plan elements for hitting something on its left side or its right, or have some state indicating which side it hit something on, and a single plan element whose primitive uses this state to decide whether to move left or right. The complexity of the information the bodbot required — and the need for persistence of data — meant that the need for extra state usually prevailed in this case.

4.2 The Primitives

This section illustrates part of the development process by presenting an example of a sensory primitive. In total, I coded 20 actions and 23 senses, and re-used the 5 actions and 9 senses of the poshbot ([14]). The sense shown in this section, *reachable-nav-point*, was chosen with a view to demonstrating interesting features of the bot, such as its use of state, the trade-offs between plans and behaviours and so on. For ease of explanation, I have broken it up into sections:

```
# returns True if there's a reachable nav point
# in the bot's list which we're not already at
def reachable_nav_point(self):
    # setup location tuple
    if not self.bot.botinfo.has_key("Location"):
        # if we don't know where we are, treat it as
        # (0,0,0) as that will just mean we go to the
        # nav point even if we're close by
        (SX, SY, SZ) = (0, 0, 0)
    else:
        (SX, SY, SZ) = utilityfns.location_string_to_tuple(
                                self.bot.botinfo["Location"])
```

As part of this sense, we must already determine whether we are already close to the navpoint we are aiming for. Our location is stored in the *botinfo* dictionary. However, this is stored as a string and thus must be converted into a tuple (in this case, a triple) for comparison, hence the call to *utilityfns.location_string_to_tuple*. This line also provides an example of Python's ability to perform multiple-assignment.

If the location is not available, we can treat the bot as being at *(0,0,0)*. This might mean that we are actually close to a navpoint but do not realise it, but it is worth taking this minor risk rather than doing nothing.

```
# is there already a navpoint we're aiming for?
# how near we must be to be thought of as at the nav point
DistanceTolerance = 30
if self.PosInfo.ChosenNavPoint != None:
    (NX, NY, NZ) = self.PosInfo.ChosenNavPoint
    if utilityfns.find_distance((NX, NY), (SX, SY)) >
                                        DistanceTolerance:
        return True
    else:  # set this NP as visited
        self.PosInfo.VisitedNavPoints.append((NX, NY, NZ))
        self.PosInfo.ChosenNavPoint = None
```

It may be that the bot has already chosen a navigation point to aim for (*self.PosInfo.ChosenNavPoint*) and is currently heading there. In this case, we test whether the bot has already got there. This uses another utility function, *find_distance*. If the bot is not already there, then we need do nothing more – the bot has a location to head for so we can simply return. However, if the bot is there then we add the point to our list of visited navpoints and clear the variable stating where we are heading for. We do not return from the function but rather continue execution to find a new navpoint.

This extract of code is an interesting one as it is an example of something which could be accomplished either in a primitive (as here) or by making the plan file more complicated (i.e. adding a sense to check whether we are at the place we're heading and an action to clear it if we are.) There is no overwhelming advantage to either method, it is more a matter of personal preference. The trade-off this demonstrates (between complexity of plans and complexity of primitives) is an important one, however.

```
# now look at the list of
# navpoints the bot can see
if self.bot.nav_points == None or
                          len(self.bot.nav_points) == 0:
    return False
```

If the bot cannot see any navpoints then the sense obviously fails.

```
else:
    # nav_points is a list of tuples.  Each tuple
    # contains an ID and a dictionary of
    # attributes as defined in the API
    # Search for reachable nav points
    PossibleNPs = self.get_reachable_nav_points(
```

```
            self.bot.nav_points.items(),
            DistanceTolerance, (SX, SY, SZ))
```

The *get_reachable_nav_points* function takes a list of navpoints and returns a list of all those which are specified as "reachable" and which the bot is more than *Distance-Tolerance* units away from[2].

```
# now work through this list of NavPoints
# until we find one that we haven't been to
# or the one we've been to least often
if len(PossibleNPs) == 0:
    return False # nothing found
else:
    self.PosInfo.ChosenNavPoint =
        self.get_least_visited_navpoint(PossibleNPs)
    return True
```

The function now searches this returned list (unless it is empty) and finds the one visited least often. This is accomplished by the *get_least_visited_navpoint* function which searches the list in *self.PosInfo.VisitedNavPoints*.

self.PosInfo.ChosenNavPoint is set to this least-visited navpoint. This variable then used by the *walk-to-nav-point* action primitive to actually make the agent run to this navpoint.

5 Conclusion

The final agent was one of the most complex BOD agents yet published (see further Partington [17]).

We found that BOD offered the following key advantages:

- More focussed development. Because an Action Selection mechanism was provided it did not need to be coded.
- An ease in constructing goal parallelism. This allowed both for higher-priority drives to interrupt lower-priority ones, and two goals to be pursued at once.
- The ability to set frequencies for pursuing goals and retries limits for attempting actions. This made fine-tuning of the agent's action selection relatively easy.

A number of minor problems with both pyPOSH and the methodology were discovered, some of which have already been addressed in the course of this project. Others will need to be addressed as future work. In particular, it would be useful to have a full-blown interactive development environment for debugging POSH plans.

Some problems in agent development are still just hard, particularly navigation and debugging the Gamebots interface itself. There is no way around needing to make elaborate modules for these sorts of problems. However, the fact that they *are* modules, and can be treated distinct from other problems, did at least simplify their construction. In general, we strongly recommend the BOD methodology.

[2] "Units" refers to Unreal Tournament distance units, discussed in the Gamebots API

References

[1] Ronald C. Arkin. *Behavior-Based Robotics*. MIT Press, Cambridge, MA, 1998.

[2] Christian Balkenius. *Natural Intelligence in Artificial Creatures*. PhD thesis, Lund University Cognitive Studies, 1995.

[3] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 2000.

[4] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2:14–23, April 1986.

[5] Rodney A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47: 139–159, 1991.

[6] Joanna J. Bryson. Cross-paradigm analysis of autonomous agent architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 12(2):165–190, 2000.

[7] Joanna J. Bryson. *Intelligence by Design: Principles of Modularity and Coordination for Engineering Complex Adaptive Agents*. PhD thesis, MIT, Department of EECS, Cambridge, MA, June 2001. AI Technical Report 2001-003.

[8] Joanna J. Bryson. The behavior-oriented design of modular agent intelligence. In R. Kowalszyk, Jörg P. Müller, H. Tianfield, and R. Unland, editors, *Agent Technologies, Infrastructures, Tools, and Applications for e-Services*, pages 61–76. Springer, 2003.

[9] Joanna J. Bryson and Lynn Andrea Stein. Architectures and idioms: Making progress in agent design. In C. Castelfranchi and Y. Lespérance, editors, *The Seventh International Workshop on Agent Theories, Architectures, and Languages (ATAL2000)*. Springer, 2001.

[10] Peter Coad, David North, and Mark Mayfield. *Object Models: Strategies, Patterns and Applications*. Prentice Hall, 2nd edition, 1997.

[11] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society series B*, 39:1–38, 1977.

[12] Epic Games. Unreal tournament, 2004. http://www.unrealtournament.com/utgoty/, Accessed 2 November 2004.

[13] G. A. Kaminka, M. M. Veloso, S. Schaffer, C. Sollitto, R. Adobbati, Marshall, A. N., A. Scholer, and S. Tejada. GameBots: A flexible test bed for multiagent team research. *Communications of the ACM*, 45(1):43–45, 2002.

[14] Andy Kwong. A framework for reactive intelligence through agile component-based behaviors. Master's thesis, University of Bath, 2003. Department of Computer Science.

[15] Pattie Maes. Situated agents can have goals. In Pattie Maes, editor, *Designing Autonomous Agents : Theory and Practice from Biology to Engineering and back*, pages 49–70. MIT Press, Cambridge, MA, 1990.

[16] David Lorge Parnas, Paul C. Clements, and David M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3): 259–266, March 1985.

[17] Samuel J. Partington. A critical analysis of behaviour-oriented design (BOD), based on experiences in using it to create an unreal tournament capture-the-flag (CTF) team, *expected* 2005. Undergraduate Dissertation, University of Bath.