Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

Starcraft Hacking Recognition

Starcraft csalásdetektálás

Author: András Belicza
E-mail: iczaaa@gmail.com

Consultant: Károly Kondorosi, dr.

2009

Kivonat

A diplomaterv témája napjaink egyik népszerű video-játékához, a Starcraft-hoz egy csalásdetektáló eszköz kifejlesztése.

A csalás egyre nagyobb problémát jelent a Starcraft játékban, mivel egyre több játékos folyamodik csaláshoz. Csalással védekezni a csalók ellen nem éppen ésszerű megoldás, ugyanis a csalás törvénytelennek minősül és elveszi a játék igazi élményét. A legtöbb amit tehetünk, hogy elkerüljük a csalókat. Ehhez persze arra lenne szükség, hogy tudjuk, kik csalnak. Egy ezt támogató rendszer minimális képessége az, hogy legalább a játékok után tudatja velük, hogy kik csaltak a játék alatt. A legjobb megoldás az lenne, ha már a játék előtt meg tudnánk mondani, hogy vannak-e ismert csalók játékos társaink között. Így mi dönthetnék arról, hogy mit teszünk ez esetben.

A mi megoldásunk, a BWHF Agent program ezt törvényes módon éri el: egy olyan rendszert épít fel, mely a játékok után képes megmondani, hogy kik csaltak, és ha rendelkezünk egy erre felhatalmazó kulccsal, automatikusan jelenteni tudja a talált csalókat egy központi adatbázisba. A központi adatbázisba jelentett csalók aztán ellenőrizhetők, hogy csatlakoztak-e a mi játékunkba. Mi dönthetjük el, hogy kitiltjuk őket a játékból, elhagyjuk a játékot vagy pedig egyszerűen nem teszünk semmit. Mindeközben a BWHF Agent egy törvényes program, nem csak egy másik "jóindulatú hack program", mert nem sérti a játék készítőjének, a Blizzard-nak a felhasználói szabályzatát.

A diplomaterv első fejezete bemutatja a Starcraft játékot. Megismerjük a játék elemeit, megtudjuk, hogy mit tart a kezében a játékos, mi múlik a játékosok ügyességén. Bemutatja a tipikus csalási módokat, hogy mi automatizálható programokkal, és hogy milyen előnyre tesznek szert a csalók.

A második fejezet a rendszer architektúráját és technológiai megoldásait ismerteti, és kitér a jelentéseket küldő felhasználók azonosítására.

A 3. fejezet a szerver architektúráját és implementációját részletezi. Betekintést nyerünk az adatbázis szerkezetébe, megismerjük a kliens programok kiszolgálását. A központi szervernek van egy web interfésze is, ahol a korábban jelentett csalókat listázhatjuk, különböző statisztikákat nézhetünk meg.

A következő fejezet a kliens programmal foglalkozik, elsősorban hogy hogyan ismeri fel a csalásokat, és hogyan képes észlelni és jelezni a már korábban jelentett csalók jelenlétét. Mivel a rendszer igényel egy, a felhasználók gépén futó kliens programot, számos más, a játékosok számára hasznos feladatot is el tudunk látni. Megismerjük ezeket és a megoldások módszerét.

A BWHF Agent egy hosszú távú projekt. Az 5. fejezet a projekt fejlesztési módszertanát taglalja, és azt, hogy a felhasználók ebben milyen szerepet játszanak.

A 6. fejezet a projekt történetét foglalja össze: mi vezetett a rendszer kialakulásához, mik az elért eredmények és hogyan lehet még tovább fejleszteni a projektet.

Abstract

The subject of this diploma work is to design and implement a hacking recognition system for one of today's popular video game: Starcraft.
Hacking (cheating) is becoming a bigger and bigger problem in Starcraft as more and more people uses hacks. Fighting back with hacks is not the right solution: using hacks is illegal and it ruins the gaming experience. The best we can do is to avoid them. For this we would have to know who hacks. A system aiding this would have to at least inform us of the hackers found in the last game. It would be the best solution, if it could alert us before the game starts if presence of known hackers is detected. That way we could decide what to do about it.
Our solution, the BWHF Agent achieves this in a legit way: it builds a system which can detect hacks after games, and with the possession of authorization keys it reports detected hackers automatically to a central hacker database. Then the reported hackers in the database can be checked if they are in the same game with us. The decision is in our hands whether to kick/ban them, leave the game or go along with it. Meanwhile BWHF Agent remains legit: it is not just a well-intentioned hack itself, but it does all this without violating Blizzard's (the creator of Starcraft) terms of use.

The first chapter of this Thesis will introduce the Starcraft game. We get to know the elements of the game, and what is under the control of the players and what depends on the skills of the players. It describes the typical hacking methods, what can be automated and done with hacks, and what is the gain of using hacks.
Chapter 2 describes the architecture of our hack detection system, what technologies it uses and how authorization keys ensure us the legitimacy of the reports.
The 3rd chapter presents how the central hacker database works: how it stores the hackers and how it serves the client requests. It also has a web interface where the reported hackers can be browsed and different statistics can be viewed.
The next chapter explains how the client works, how it detects hacks and how it recognizes the presence of previously reported hackers. Being a client application ran on the players' computers we can perform a lot of other useful tasks. We will get an insight of what these are and how they are implemented.
BWHF Agent is a long term project. Chapter 5 explains the development methods and how users and their feedback shape the development of the project.
Chapter 6 reviews the history of the project: what led to this system, what we achieved and what are the possibilities to improve the system.

Contents

1 Introduction

1.1 Introducing Starcraft

Starcraft is an 11-year old real-time strategy video game developed by Blizzard Entertainment. It was first released in March 1998 for Microsoft Windows, and a MAC OS X version was released a year later. It has become one of the best-selling games ever for the personal computer with more than 10 million copies sold worldwide.
Based on the success of Starcraft several expansion packs have been released. The most important one was at the end of 1998 called Starcraft: Broodwar continuing the storyline of Starcraft and introducing new elements to the game.

Many of the industry's journalists have praised Starcraft as one of the best and most popular video game of all time. Starcraft is incredibly popular in South Korea, it is basically their national e-sport. Tournaments are regularly held there where professional players and teams earn sponsorship and games are broadcasted on televisions and on the internet.
Starcraft is gaining popularity even nowadays. An official Starcraft class began this semester at UC Berkley. This exciting new class orients around Game Theory with applications to Starcraft.

The game has massive multiplayer support including players up to 8 in a game. Games can be played through direct cable connection, on LAN networks with IPX/SPX and UDP protocols and over the Internet connected through official or custom so called Battle.net servers. Even today the number of players using Starcraft Broodwar only on the official battle.net servers is around a hundred thousand at every moment.

1.1.1 Elements of the game

Players have buildings and units to control.
Disregarding custom and use map settings, by default our goal is to destroy all the opponents' buildings. In order to do this, we have 2 kinds of resources to manage: vespene *gas* and *minerals*. Minerals are needed for the construction of every unit and building, gas is for advanced units, advanced buildings and for technology advance. Resources can be obtained by using *worker* units to gather them from mineral patches and vespene geysers. We decide what buildings to build, what units to train and what to do with them, what special abilities to research in order to improve units and/or buildings.
The game and battles take place on the *map*. The map is under the *fog of war*, it's black. We only see parts of it where we are present. Each unit and building has its own range of sight. The map can be assembled from numerous elements: different types of grounds, highlands and water. There can

be big spaces to move or build as we like, or barrials, passages, ramps or areas which we cannot cross with ground units or we cannot build on. Units on the highlands usually have advantage over the ones being on normal grounds.

Starcraft has 3 completely unique *races*: Protoss, Zerg and Terran. This is how wikipedia.org describes these races:

> *"The enigmatic **Protoss** have access to powerful units and machinery and advanced technologies such as energy shields and localised warp capabilities, powered by their psionic traits. However, their forces are slow and expensive to produce, encouraging players to follow a strategy of the quality of their units over the quantity. The insectoid **Zerg** possess entirely organic units and structures, which can be produced quickly and at a far cheaper cost to resources, but are accordingly weaker, relying on sheer numbers and speed to overwhelm enemies. The Zerg augment their forces through evolution, developing armoured carapaces and various types of claws, spines and acids as weapons. The **Terrans** provide a middle ground between the other two races, providing units that are versatile and flexible. The Terrans have access to a range of more ballistic military technologies and machinery, such as tanks and nuclear weapons. Although each race is unique in its composition, no race has an innate advantage over the other. Each species is balanced out so that while they have different strengths, powers, and abilities their overall strength is the same. The balance stays complete via infrequent patches provided from Blizzard."*

Each race has its own buildings, units, researches and own *dependency graphs*. The dependency graph describes what units/buildings/researches can be made based on what we have, and describes what is missing to complete an operation. The dependency graphs are static, and even though we do not have to learn them by heart because if we cannot do an operation due to unsatisfied conditions we see what they are, on high level games there is no time to fix if we are not prepared. An example: we cannot train *medic* units from the *Barracks* building if we do not have an *Academy* building. These differences between races evolved to completely different strategies to be used and be successful for each.
We can see a Protoss force attacking a Zerg colony in the screenshot below:

1.1.1.1 Units

The unit model is very complex in Starcraft. There are 2 basic types of units: *ground* and *air*. Air units basically can fly everywhere, the movement of ground units is restricted by the different ground elements, buildings and other ground units.

Every unit has its general properties such as *minerals* and *gas* it costs to build, *hit points* and *shield* (in case of the Protoss race), *build time*, *attach range* and *sight range*. The damage they do might differ to ground and to air units.

The damage system is more complex. As mentioned, every unit has a property of *ground attack* and *air attack*. Moreover units has an *armor* property: the damage a unit can make will be reduced with the amount of the armor of the target unit. At first sight. But units has also a *unit size* property which as values of *small*, *medium* and *large*. The damage also depends on the size of the target unit, for example on the half of the damage is carried out if a *small* unit is targeted. Also, some units has *splash* damage which means it hits not just the targeted units but also units being *near* to the target.

Units may or may not have special *abilities*. Special abilities and their effects are completely unique. Some slow down targeted units (*ensnare*), some damages *biological* units (*irradiate*), some might damage all targeted units (*psionic storm*), some might reduce or take away all protoss shields and *energy* (*EMP shockwave*), some might block targeted units (*lockdown* and *maelstorm*). There

are a lot more. Abilities might be natural abilities of units, some might require researches. Abilities uses the *energy* properties of the units. Energy charges over time.

Armors, shield and attacks can be upgraded 3 times. If upgraded, a small amount is added to the appropriate properties of the concerned units. Upgrades effect all present and future units. At first glance, upgrades might not have big significance. For example the unit *Zealot* has 16 ground attack. If upgraded once, it gets another 2 (18). Upgrades take time and special building where we can upgrade certain things. Examining the numbers from closer, upgrading the right property deals a great technological advance.

For example if we are Protoss and the opponent is Zerg, and we *know* (thanks to the continuous scout) that he is making lots of *zergling* units, we can make lots of Zealots and upgrade the ground attack. Zerglings have 35 hit points, zealots have 16 ground attack which means a zealot has to hit a zergling 3 times to kill it (16+16<35 but 16+16+16>=35). However if we upgrade the ground attack, the zealot will have 18 ground attack which makes it only **2 hits** to kill a zergling (18+18>=35). By upgrading, we basically gained 33% more hit power.

Of course every upgrade has its *counter* upgrade. If in the previous case the Zerg also upgrades his ground armor, that will make every attack point one less, in this case 17 (instead of 18). And 17+17 is only 34 which is less that 35, so a zealot will have to hit 3 times again to kill a zergling.

1.1.1.2 Skills and strategies

Throughout the 11 years of Starcraft the following different skills have been developed and are distinguished during a game:

- **Micro:** this element of the game means controlling units in a battle to fight as efficient as possible. The formation of the units, their combination, their movement during the fight, positioning on low and high grounds and timing all matter.
- **Macro:** this element refers to the production of units and buildings. Decisions to what buildings and units to make, and to make them continuously and periodically to use the available resources to the best. Macro can be improved by placing buildings in formations such as lines or matrices so we can select them train units faster. Buildings usually support up to 5 units to be queued in them for production. However doing this we have to pay all the resources for the queued units but on the other hand we do not have to return to this building each time when a unit is ready to build the next one. Ideal would be to always train the next unit when the previous is ready.
- **Strategy:** to choose the good or best strategy (what, when and how to build, and what to do with them) it involves a lot of things. It depends on the map, the races, how previous strategies and battles turned out, the skills of the players. Every strategy and unit has a counterpart. There is no absolute strategy that would lead to victory regardless to the

opponents strategy. Strategies might change throughout the game, especially if we know what our opponent is doing. Exploring the opponent is very important. Some elements of different strategies:

- **Scout:** this refers to exploring and scouting the map and what the opponent does.
- **Expand:** to build a new base in order to be able to gather minerals from another location. Takes some time and resources, but it has its benefits on the long run of course.
- **Harass:** to bother the opponent in some ways, the goal is to delay the opponent, to make him/her lose focus on what he/she was or is doing.
- **Drop:** to create transporter units, and drop ground units back in the opponents base usually to kill his economy
- **Rush:** to make units fast (or faster than expected) and do an early attack.
- **Proxy:** to construct buildings not in our base, but hidden in the opponent base (or close to it in an unexpected location), so we can make units pop out much closer to the opponent. This can be dangerous if the opponent does not count on it, but can be a loss if this strategy fails and the player loses the proxy buildings.
- **Tech:** this refers to skip making basic units (or just a few of them) and rather go for a fast technology advance.
- **Mass:** this refers to making a lot of units, massing them rather than going significantly advanced units and technologies.

Good multitasking and fast decision making ability is a must for a good player to do all these parallel and at the same time.

1.1.2 Communication between players

A Starcraft game played on the network is client side computed meaning the players only share the actions the players do during a game, everything else is computed on all participants' computer. The basic unit of the game execution/calculation is the *iteration* or *frame*: there are approximately 24 iterations in every second (1000/42 precisely). Actions happening between iterations are put in a queue and will be executed when the next iteration is activated.

To handle network delays the game provides a latency setting. This value specifies the interval of time while actions are gathered from the player (thorough several iterations), and will be distributed at the end of this time. Every action made by the players will be delayed with the length of this interval. When temporal network error occurs, the game is put in a waiting state. However, some kind of pile up in the number of actions might happen. We have to handle this during hack detection.

1.1.3 Replays

Years after Starcraft has been released, a patch came out allowing players to save games after the game has ended, and a later patch even made the game automatically save all games that are played to a specific file (the previous game is always overwritten). We will use this feature in our system. These so called *replays* contain all the information required to re-play the game: the map, the players as participants, and all the actions the players gave during the game.

1.2 Hacking, and the process and possibilities of hack detection

The definition of hacking in Starcraft terminology is: doing *something* which results in **gaining an unfair advantage** compared to those who play legit.

Hacking is becoming bigger and bigger problem as newer, better and more hack programs are made. Many of them are undetectable while at the same time giving more and more advantage to the hacker. Many people hack just because they think everyone else do and they cannot be competitive without hacking themselves.

Hacking has 2 forms:

1. We run a 3rd party program which interacts with Starcraft: reads and writes the memory of Starcraft which results in something we could not have achieved without that 3rd party program no matter what we would do inside Starcraft.
2. The other type is based on exploiting bugs in Starcraft. This type is called *cheating* rather than *hacking*, because this method does not require any 3rd party hack program.

Only the first type is forbidden and considered illegal. Blizzard's terms of usage forbids all types of 3rd party programs falling into the first type.
Example for the cheating: the terran *vulture* units can place spider mines on the map. Enemy units walking close them will activate them, and the mines will be driven to the enemy units and explode. However, if the player first allies the opponent, the spider mines will not be activated. The opponent will walk onto a field full of spider mines. Once the opponent's army is on the field, the player unallies and all spider mine will be activated at once.

In what follows I will only deal with the first type.

A subset of hacks builds on the fact that Starcraft is not fully specified. By that I mean the following: certain actions might be valid and meaningful, if they're given to specific units. A building command to a non-builder unit should be invalid and meaningless. Starcraft itself does not

allow sending such invalid commands. And based on that it does not always check the received commands. It will execute the received commands of others without checking if it is valid. Executing a meaningless command usually results in crash or some player being dropped which ends the game. But some might actually result in a state which gives the player an unfair advantage. Let's call these hacks *exploit*s.

1.2.1 Types of hacks

The main types of hacks and their gain:

- Maphack: This hack reveals the whole map and all objects on it which is already on the players' computers because it is required for game computing.
- Mineralhack or moneyhack: This *exploit* gives a specific amount of gas and/or minerals to the hacker without earning or gathering it.
- Multicommand hack: This hack is basically what the player could do by himself, just automates it and does it faster and more precise. Subtypes of multicommand hack:
- Autogather/autotrain: automatically makes new units and automatically sends them to gather resources when they are ready.
    - Building selection hack: allows multiple buildings to be selected at a time which is not possible and allowed in Starcraft. The gain of this hack is that the player only needs to give one command, and that command will be executed on all selected buildings.
    - Multicommand rally set hack: allows the hacker to set the rally point of all selected buildings to the exact same location.
    - Multicommand unit control hack: Starcraft only allows 12 units to be selected at a time, and they can be controlled as a group of 12. This hack virtually allows the hacker to select more than 12 units at a time (even 100 for example), and commands given to this group will be executed to all the selected units. The hack is responsible to map the given command to smaller groups (units of 12 at the most).
- Drophack: This *exploit* causes some or all of the players to be dropped out of the game.
- Stack hack: This *exploit* allowed the hacker to build several buildings on top of each other which was not allowed in Starcraft.
- Name spoofing: This *exploit* allows the hacker to appear under any name he wishes. With this hack, one can hide his true identity.
- Host hack: This *exploit* allows the hacker to operate as the game operator. Allows the user to give commands in the game lobby which he does not have privileges.

1.2.2 History of hack detection

Recognizing a hack action or a hacker is not always an exact fact. There might be actions or player behavior which points to hack, but lot of hacks are undetectable. My principle is to rather let a hacker go than to stamp an innocent player. I always make every effort to be a hundred percent sure of alerting for hack, which means that no matter what the player would have done, he/she could not have achieved the results without using a 3rd party hack program. On the other hand, if we have found 1 proof of hack, that is enough: no matter how many times a person hacks, a hacker is a hacker.

There are 2 basic types of detection:

- Online: we detect the hacks during the game
- Offline: we detect the usage of hacks after games by analyzing the replays

Online detectors are actually hacks themselves, because detecting the presence of hacks online requires reading Starcraft's memory, which is part of the hack definition. They are forbidden by Blizzard.

The only allowed online detector is called iCCup launcher. This program runs along with Starcraft, and checks certain offsets of Starcraft if they were modified by other programs. Of course it is a program, it can be hacked just like any other (meaning that I can run the program showing that I don't hack and at the same time hack the antihack program and still hack).

A very famous old online hack detector was the Penguin Plug. It was able to detect most of the hacks that existed at that time: even maphack. Its algorithm was to warn the user if someone selected an object which was not visible at the time of selection. It was quite useful regardless the fact that it required human revision because in some cases it alerted even if it was not truly a hack action.

Then *client-side safe* maphack was created. The point of this is that the hack program makes everything visible to the user, but only at his computer. If he selects objects that he cannot see, those actions are only handled by the hack program, and are not distributed to the others. In fact, the hack creators wanted to be extra safe, so they intentionally removed all select actions which targeted objects not owned by the hack user. This was a mistake on their part. A new algorithm surfaced where we could look for mass of replays where someone had absolutely no actions selecting others units (because statistically this happens for normal users quite often). But for those who used this hack, the hack itself removed these actions. Of course this was fixed soon too. Now again a new method surfaced for detecting maphacks: a new thing which was considered a feature introduced by today's maphacks: *build anywhere* hack. This feature allowed the hack users to give the build command targeted to anywhere… even where Starcraft itself does not allow building. These actions can be reviewed and serve as proof of using maphack. For example if the player tries to build a unit at the bottom of the map, but if there is not enough room for it, Starcraft does not

allow this action, therefore it will not be distributed and recorded in replays. The hack itself however overrides this, and allows the command to be sent out – even if it will not be executed eventually. In the end the fact that this action is recorded in the replay proves that the player was using hacks.

At the time of writing this thesis, there is no currently working online detector what would check for actions and patters for hacks. The closest one that was working in the past was called Chaosplugin, but these online detectors are bound to a specific Starcraft version, and they stop working after Starcraft updates. The same thing happened to Chaosplugin, and after that it was not updated.

There are some other offline replay scanners which were targeted for a specific, easily recognizable hack. They are very simple, very limited in usability, and without proper GUI they are often difficult to use.

There is a program called BWChart, which has excellent GUI, but the main goal of this program is not hack detection. It is used for organizing and analyzing replays. It has very limited hack detection feature (and only 1 replay at a time) and the development of this program was discontinued several years ago.

So client-side safe maphacks basically cannot be detected. Moneyhacks are easy to detect: since they are *exploits*, we usually just have to look for actions which are not valid.

Multicommand hacks though use valid commands, but they do it fast and precise: usually following certain patterns. If these patters cannot be done by a human, they can be looked at as proof of hacking.

1.2.3 Mechanism of our detector, pros and cons

My goal is to create a legit hack detector system. So it has to be an offline detector, we will have to analyze replays. And I always try to be 100 % sure of hacking.

This hack detector system is an open source project available under the GNU General Public License version 3, and it goes by the name *BWHF*. Its home page is: http://code.google.com/p/bwhf/

BWHF is a…

- …replay chart and analyzer tool for Starcraft Broodwar
- …Starcraft Broodwar hacker finder java utility library (this is the control part of the hack scanner in the MVC architecture; part of the client program)
- …client agent program called *BWHF Agent* with a purpose to search for hackers and report them (this is the client part)
- …a central hacker database (this is the server part)

The client program can be run with or without Starcraft to scan replays offline. Since Starcraft automatically saves all games when they end, we can detect this event without reading Starcraft's memory. We monitor this file named LastReplay.rep, and when its last modified date changes, we know a game has just ended, and we can perform a scan automatically. Additionally we can make a copy of LastReplay.rep, so the copy will not be overwritten when next game is saved to this file again. This feature is called *replay autosave*.

So basically the client program runs in the background and automatically checks the last replays of Starcraft for hacks. It logs the results and if it finds a hacker in the last replay it can bring its window to front or play an alert sound. It can be used as a launcher too. We can run a check on any of our saved replays or on entire folders of replays recursively.

It has some other useful features such as player checker, chart visualizer, PCX screenshot auto-converter, game chat extractor and replay search. These can help analyzing replays and finding hackers. Some of them are just useful for Starcraft players.

Those who possess a valid authorization key can report hackers automatically to the central hacker database.

Pros compared to other launchers and detectors:

- **This is not a 3rd party program!** It does not modify Starcraft's memory and it does not interfere with Starcraft in any way so there is **absolutely no risk** of getting banned by Blizzard or any battle.net server for using it.
- Since it does not modify Starcraft's memory, it does not add new bugs and does not cause Starcraft to crash.
- It is a standalone program not some kind of dll or bw plugin, so it can be used ANYWHERE (iCCup, normal battle.net servers, LAN w/e) with or without other launchers. BWHF Agent is a launcher itself.
- It can be started and closed at any time (before or after Starcraft, before or after logging in to battle.net, before or during a game)
- Players in the game lobby can be checked BEFORE game whether they have been reported to the BWHF hacker database earlier.
- It does not belong to a specific Starcraft version therefore it works after Starcraft updates without modification.
- It can be used on multiple platforms.
- The scan process can be repeated, and separate, old replays can be scanned too.
- Entire folder of replays can be scanned recursively.
- Does not require Starcraft or any other programs (like BWChart) to be running or even being installed.

- It can report hackers automatically to a central hacker database.
- It is open source therefore the risk of being/containing a malware/adware/virus is minimized (if one always downloads it from its original home page).

Cons:

- Scan runs after games (it will not tell if someone is hacking during the game).
- If reporting hacker is enabled, manual change of gateway is required if the player changes gateway (else the reports that get sent will contain the incorrect gateway).

2 System architectural design

The detected and reported hackers are maintained in one centralized place. This is the *central hacker database.* It has a web interface where everyone can list, search, filter, sort existing hackers. This web interface is available here: http://94.199.240.39/hackerdb/hackers. There has to be a client program, an *agent* program that is run by the players on their computer which listens to the automatically saved last replays and performs the scan operation. In case of hacker detection, it can contact the *central hacker database* in order to report the hackers found.

2.1 Technologies

Key points of the system

- We want it be legit by Blizzard's Terms of Use. Therefore the agent will be an independent program from Starcraft. We will not need to "hack" into the Starcraft program.
- It would be nice if the program would work on multiple platforms without much effort.

Several technologies could be selected to create this system. My choice falls on Java for the following reasons:

- I have years of experience with Java.
- Java has support for each of our areas: desktop, client applications, web servers, database servers, and it has strong network support.
- It will be ready as-is for cross platform usage.

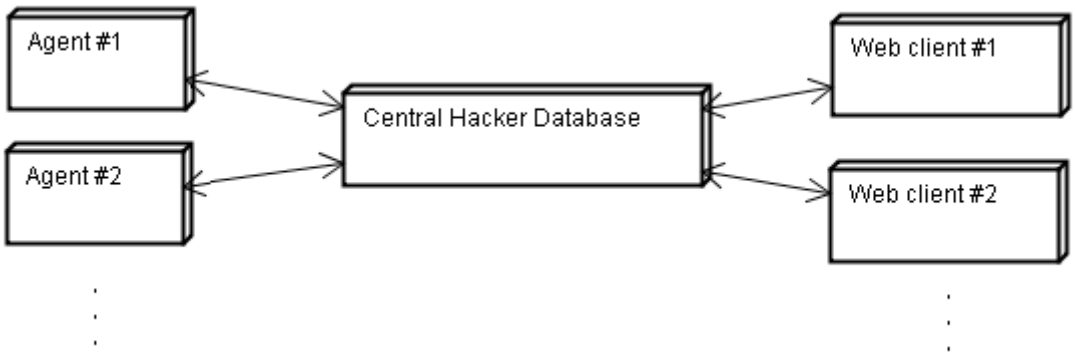
Java SE 6.0 will be used for the client program (older releases would do just as well, but I want to use the features introduced in the newer ones).

The server part will be implemented using the Java Servlet API and technology.

For development environment, I used Eclipse.

2.2 System architecture and components

The overview architectural design gives us the following look of the system:



The *Agents* are the client programs part of the *BWHF* system. The *Web clients* are any browsers that use standard HTTP protocol for the communication. For simplicity the agents will also use HTTP protocol to communicate with the server, and the parameters and information of the requests are defined in a common interface, in a common *class*.

2.3 Authorization key system

2.3.1 What is this key system?

Since the reports come from untrusted clients, we have to minimize the risk of getting false reports in order to obtain a legitimate and trustable database of hackers. I will initiate a key system for the following purposes:

- To filter out bad intentioned reports.
- To be able to identify the origins of the reports in order to filter them out or invalidate them later if the source becomes unreliable.

The key authenticates its owner toward the BWHF hacker database server. It is like the CD key for Starcraft: one needs a valid CD key which authenticates him/her toward Blizzard's battle.net servers.

Only those who possess a valid key can report hackers to the central hacker database. Reports with invalid key are discarded.

2.3.2 How can someone get a key?

Keys cannot be purchased. I give keys to those who can be trusted. This includes my friends and people I play with on battle.net. Of course this is a very limited group of people.

Moreover I provide keys upon requests to admins of known Starcraft websites and communities. Once I verified the requestor is truly the admin he/she claims to be, I provide as many keys as needed. Then the admins are responsible to distribute the keys to people they know and whom they trust.

2.3.3 Key revocation

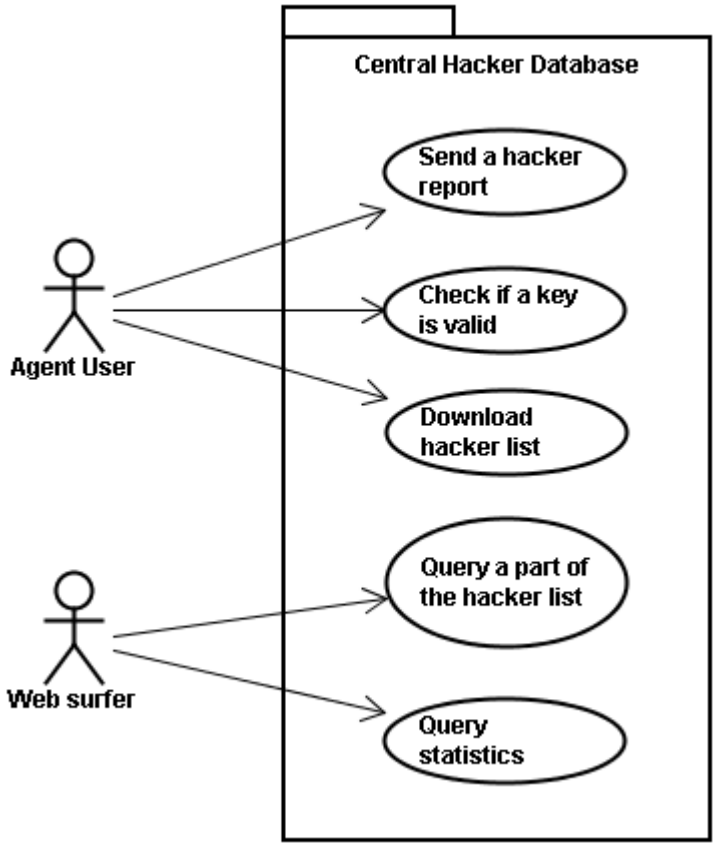
If I find that a key is used inappropriately, the key can be revocated at any time without any warning. Revocation of a key means all reports that were done using that key will be excluded from all queries like they were never happened. Optionally those reports will be deleted forever.

3 Server architecture

The server has to provide the following services:

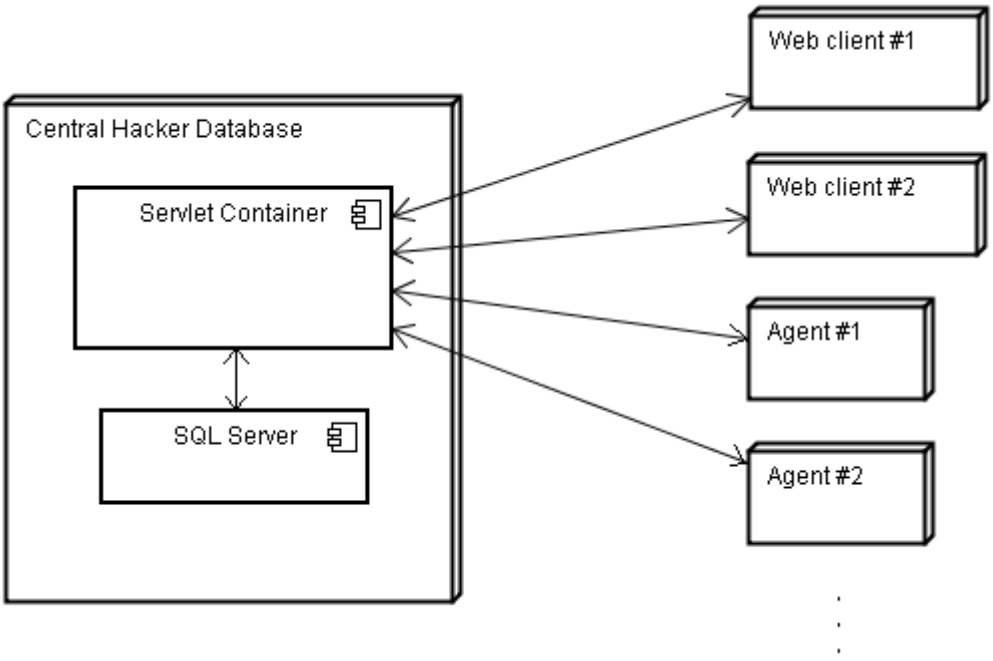
- **Manage the authorization keys and the owners of the keys** and provide interface to check if a key is valid.
- **Manage the database of reported hackers.** This basically means to store the list of hackers in a database, and allow execution of queries on the list.
- **Accept and process hacker reports.** In cases of the report containing invalid key discard action must be taken. In cases when the report has a valid key, the report have to be added to the database. New hackers have to be added *on demand* meaning if the hacker has been reported before, then we just store the report attaching to the existing hacker; else we create the new hacker identifier and attach the report to this new one.
- **Accept and serve HTTP requests coming from web clients** querying a sorted and filtered subset of hackers.
- **Allow downloading of the hacker list** for local caching in order to perform player check.
- **Provide basic statistics** regarding to the reported hackers and reports.

This is the UML use case diagram of the server:



In order to simplify the server architecture and implementation while preserving the required services, we want to accept and process the reports and hacker list queries the same way. Since we

want the hacker list interface to be a web interface, we will process hacker reports via HTTP requests. Java servlets will process the hacker reports and the hacker list queries.
To easily solve the data management including storage and querying, we will use a relational database server. The servlets will connect to the database server through JDBC interface.
Now this is how the architecture our central database server looks like:



Several database servers and servlet containers could be selected for this task. I choose the Java HSQLDB and the Tomcat 6.0 servlet container because they are standards, free, open source, easy to setup and configure, and they use quite low resources. Perfect for our server.

3.1 Database

As mentioned earlier, I will use HSQLDB. When defining the types of columns, occasionally I use special types of HSQLDB to minimize the work and effort handling the tables. An example for this is the use of the HSQLDB IDENTITY type which is auto-incremented and is treated as the primary key for the table by default.
We want the following entities stored in the database:

- The **valid authorization keys** given to the users of BWHF Agent.
- The **persons** who owns the authorization keys.
- The **hackers** reported to the central hacker database by the agents.
- The **reports** that were sent by the agents containing the hackers.

Let's design the tables. What are the key concepts?

1. A person might have multiple keys due to the following reasons:
    - He might be a *key distributor*, an admin of a Starcraft site for example.

- If a key gets stolen, the key might be revocated, and the person might get a new key.

2. A hacker might get reported several times.
3. If a key gets revocated, we want all reports be undone that happened with that key. Hackers are not directly affected because they might have other reports from valid keys. But if a hacker has only reports with invalidated keys, he/she should not be displayed.

Due to *condition 1* person and key will be separate tables, and the key will reference to the person. The key table will contain a column whether the key has been revocated. With this, the revocation process will be simply changing the column value.
Due to *condition 2* the hacker and reports are stored in separate tables, and the report will reference to the hacker.
Due to *condition 3* the report will reference the key it was sent with. We can restrict any queries to the reports that reference only to valid keys.

Since a report might contain several hackers, normalization would suggest to introduce a table connecting a report to a hacker. This would increase the complexity a little. But since this is a rare case, most reports only contain 1 hacker, I decided to leave out this table, and the reports refer to the hacker directly. In cases when 1 report contains several hackers, I create multiple reports referring to the different hackers (but other properties of reports such as the key remain the same).

Pretty much that's about it. Now let's see what information we need to store from each, and what we will use them for.
Each of our tables will contain an *id* column to function as a unique key for each entity. Moreover I use a *version* column everywhere to store when the record was created.

3.1.1 Table Person

The table stores the owners of the authorization keys. Since I do not keep track of all owner, one person can have multiple keys (admins of Starcraft websites for example). I store the name of the owners and a contact email address in case of I need to contact them regarding to the BWHF system. There is a *comment* column for any additional information.
Structure of the person table:

Column name:	**Column type:**	**Column function:**
id	identity	primary key
name	varchar	name of the person
email	varchar	contact email of the person
comment	varchar	optional additional information
version	timestamp	record creation date

3.1.2 Table Key

This table stores the valid authorization keys. As mentioned earlier, keys can be revocated. We store this information in the *revocated* column. We simply set this to true if the key gets revocated. The key itself is string containing lowercased and uppercased letters and numbers. I generate keys with a typical length of 20. I added a *comment* column to this table where I can indicate reasons for revocating a key.

Structure of the key table:

Column name:	**Column type:**	**Column function:**
id	identity	primary key
value	varchar	the unique value of the authorization key
revocated	boolean	tells if this key has been revocated (in which case it is invalid)
person	int	foreign key to the owner of the key (*many-to-one* relation)
comment	varchar	reason for revocating
version	timestamp	record creation date

3.1.3 Table Hacker

This table stores the reported hackers. A hacker reported by several people is only stored once, and the reports only reference the hackers. A hacker is identified by the account name he uses to login to battle.net. Since there are several battle.net servers and the account names are not unique across the servers, we need to store the gateway too to identify a hacker. The valid gateway list can be found in the interface of the server (the client and the server must use the same values).

Structure of the hacker table:

Column name:	**Column type:**	**Column function:**
id	identity	primary key
name	varchar	account name of the hacker
gateway	int	identifier of the gateway
version	timestamp	record creation date

3.1.4 Table Report

The report table contains the valid reports received from the agents. When an agent sends a report, a valid authorization key has to be attached. If the key is invalid, the report is discarded. Multiple hackers can be included in one request. In this case all new hackers will be inserted to the hackers table, and a new report will be linked to each. The report => hacker is a *many-to-one* association. The reason for this is if we find out that someone was not hacking, we can simply delete the appropriate report record leaving the rest which store valid reports.

People can log in to battle.net using different gateways, and separate statistics will be managed for each game engines. The statistics relates to the games played and the win-loss ratio. Game engine is either the original Starcraft or the Broodwar expansion. The valid game engine list can be found in the interface of the server.

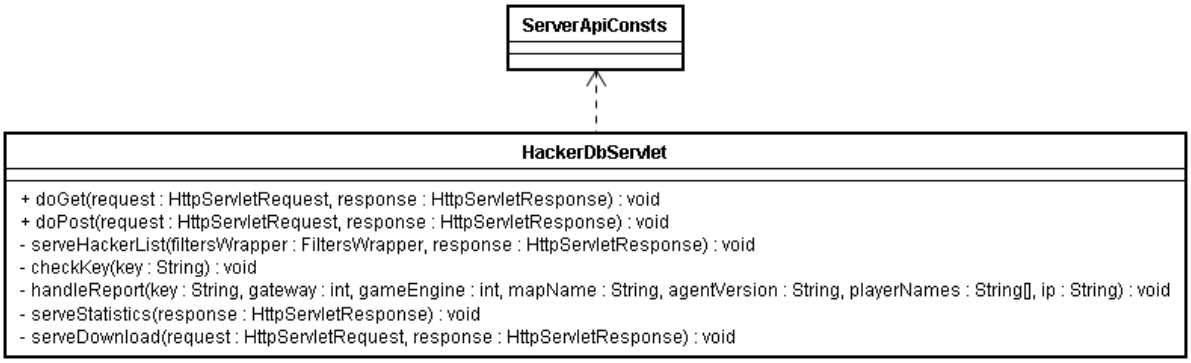
Reports store a foreign key to the authorization key which they were sent with. The gain of this is that if a key is revocated, it is stored in the key table. Knowing that we can always query reports which refers to keys only that are valid.

Structure of the report table:

Column name:	**Column type:**	**Column function:**
id	identity	primary key
hacker	int	foreign key to the hacker (*many-to-one* relation)
game_engine	int	identifier of the game engine
map_name	varchar	name of the map the game was played on
agent_version	varchar	the version of the agent that sent this report
key	int	foreign key to the authorization key which this report was sent with (*many-to-one* relation)
ip	varchar	IP address of the reporter's computer
version	timestamp	record creation date
comment	varchar	any comment regarding to the report, spoof can be indicated for example

3.2 Serving and processing client requests and reports

The client requests and reports will be processed by a servlet called the *HackerDbServlet*.



The servlet is parameterized via standard form parameters. The servlet accepts both HTTP GET and POST methods. The parameters can be either encoded in the URL, or in the body of HTTP request.

The *ServerApiConsts* class contains the name of the parameters and the possible values where the value domain is restricted. Both the HackerDbServlet itself and the clients uses this common interface for communication therefore there will be no miscommunication.

The servlet's *doPost()* method simply calls the *doGet()* method.

The *doGet()* method is responsible for parsing the parameters. If required parameters are missing or parameters have invalid values, we can decide to use default values where it is possible and preferred, or to send back an error message. The required operation is specified by the *ServerApiConsts.REQUEST_PARAMETER_NAME_OPERATION* parameter. If parameters are parsed successfully, this method delegates to the appropriate method to complete the operation.

The *handleReport()*, *serveHackerList()*, *checkKey()*, *serveStatistics()* and *serveDownload()* methods in order are the actual implementation of the previously listed use cases: *Send a hacker report*, *Query a part of the hacker list*, *Check if a key is valid*, *Query statistics* and *Download hacker list*.

The servlet connects to the database through the standard JDBC interface, therefore we could easily replace the database server. We use the JDBC driver shipping with the HSQLDB server.

3.2.1 Responses

The response of the *checkKey()* is simply the string representation of the boolean result of the check.

The response of the *handleReport()* method is *ServerApiConsts. REPORT_ACCEPTED_MESSAGE* in case of success. If the report is discarded, then an error message is sent back. This can refer to an invalid authorization key, or an internal report processing error.

The *serveDownload()* method sends the list of hackers in plain text. Each line contains a hacker, first the number of the gateway, and after that the name of the player separated with a comma.

The *serveStatistics()* method produces an HTML response where the following statistics are assembled and generated:

- **Hacker gateway distribution**: a 3D pie chart in which hackers are counted for each different gateways.
- **Monthly reports count**: a bar chart where there is a column for each month from the launch of BWHF Agent (December 2008).

On the right of each chart a table is rendered with the list of values visible on the charts. The chart data is produced by the *serveStatistics()*, and an HTML image element is inserted in the outgoing

HTML document. The charts are displayed using the Google Charts API. The charts data are encoded in the URL pointing to the Google charts server. The charts are drawn and served by the Google servers.

The *servHackerList()* method produces the list matching the filters, and generates an HTML document from it. When sending this response, we set certain fields in the response header to disable page caching. This is required because the content is dynamic and might change from seconds to seconds.
This HTML document consists of the following parts:

- **HTML page header** containing page name and CSS styles for basic formatting.
- **Page header** with the page name and links to the BWHF home page and to a help wiki page. This help page describes the elements and usage of the page.
- **Filter section** which is a table of HTML UI components to display and set the values of the filter parameters. It also contains a "Reset filters" button.
- **Go/Refresh** button to submit changes in the filters or in the following pagination section.
- **Query result field** to indicate the number of hackers matching the filters.
- **Pagination section** which allows us to break the result list into pages, to navigate between these pages, to see the current page number, the number of pages and the size of a page (the number of hackers listed on one page).
- **A page of hackers** who match the filters. This table contains summarized data for all reports referencing the same hacker.

The rendered HTML contains the current values of the filter and pagination parameters, and contains rendered *javascript* codes handling the hacker table sorting and form submit.

3.2.1.1 Filtering

The hacker database can be filtered with various properties. Only reports matching all the filters will be counted and displayed.
These are the filter fields:
A report/hacker will be counted if:

- **Hacker name:** the hacker name contains this lowercased text
- **Gateways:** the hacker belongs to one of the selected gateways
- **Map name:** the map of the replay of the report contains this lowercased text
- **Min report count:** the hacker has received at least reports of this count
- **Reported with key:** the report was done using this key; a key owner key use this to list hackers that he reported

3.2.1.2 Sorting

The hackers table can be sorted by clicking on the header of the column we want to sort by. Clicking on the same column again will change the sort order (reverse it).
The columns of the hackers table and their meaning:

- **#:** the ordinal of the hacker in the current sorting
- **Name:** the name of the hacker
- **Gateway:** the getaway of the hacker
- **Report count:** the number of reports of the hacker
- **First reported:** the date and time of when the first report of the hacker arrived; this has the meaning of how long has been the hacker hacking, or rather when was he first recorded in the database
- **Last reported:** the date and time of when the last report of the hacker arrived; this has the meaning of when was the last activity of the hacker, or rather when was he last recorded

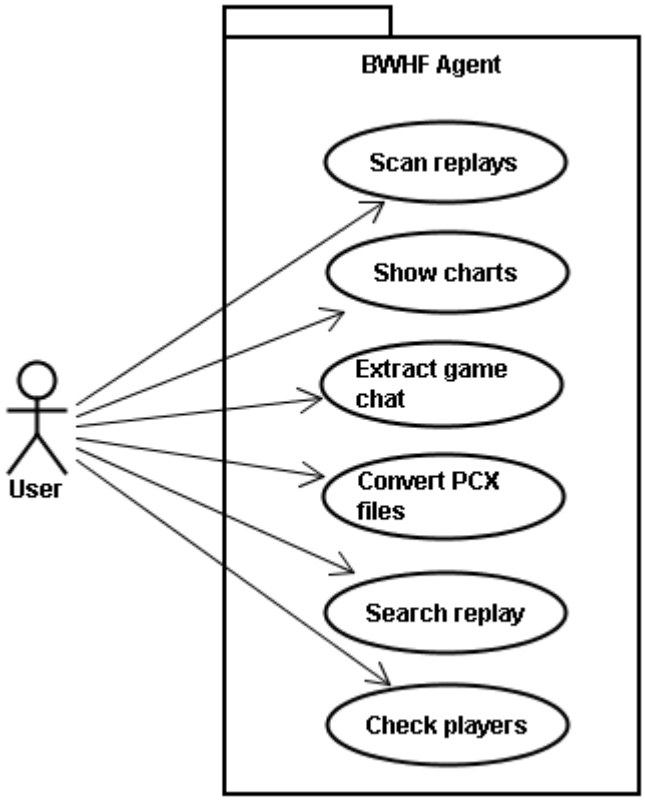
The First reported column shows how long the hacker is using hacks and the Last reported column shows how recently was the hacker active (based on the reports arrived to the BWHF system).
The default sorting column is the Last reported column, which puts the most recent active hackers to the top of the list. Sorting by the First reported column puts the newest hackers to the top of the list (based on the reports arrived to the BWHF system).

4 Client architecture

The BWHF Agent client application has the following features:

- **Listens to** the LastReplay.rep, and **automatically performs checks for hacks**. If the client has an authorization key, reports hackers automatically and alerts the user. If LastReplay.rep changes, performs replay archiving. Hacker replays can be copied to a different folder.
- Provides a way to **manually scan "old" replays or folders of replays**. Creates an HTML summary report at the end of manual scans. Hacker replays can be flagged by renaming them.
- **Shows different charts of replays**, and lists the actions recorded in them.
- **Displays or extracts in-game chat** from replays with replay header information.
- **Converts PCX screenshot images** made by Starcraft. If the user wishes so, listens to new PCX screenshots in Starcraft's directory, and automatically converts them.
- **Searches replays** based on different replay header fields.
- **Check the players** in the game lobby whether they have been reported to the BWHF hacker database earlier.

The main use cases are the following:



4.1 Technologies and frameworks used

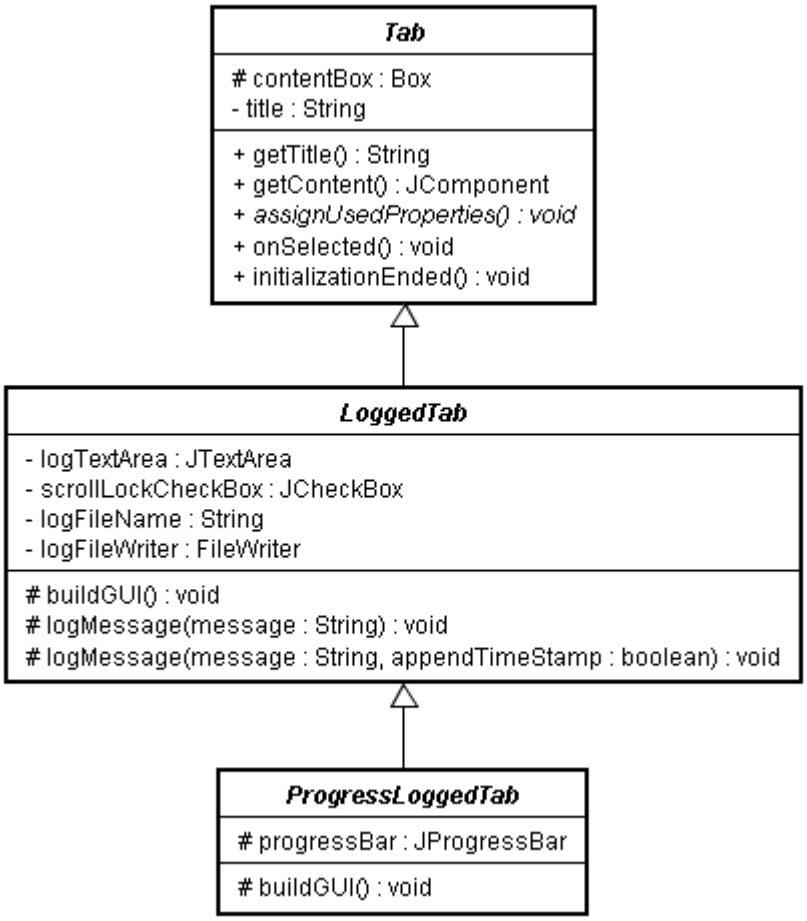
As mentioned earlier, we will use Java 6.0 SE for the client agent application. This platform supports all areas we need.

There is one problem though. Starcraft is a very old game. It runs in 640x480x256 graphics mode with DirectX. I have made several tests, and I have come to the conclusion that the Java runtime environment has a bug, some kind of garbage collection problem. When a Java GUI program is started (either AWT or Swing) and we switch to Starcraft and we start a game, the independent Java program starts to consume more and more memory. If we minimize Starcraft (switch to another application), the memory increasing stops. This applies even to the smallest Java GUI program which has nothing more than a single Frame. Meanwhile I noticed that Java applications using the Eclipse SWT framework (such as Eclipse, Azureus) work fine and this anomaly does not appear.

This forced me to use some external GUI framework. SWT would be a workable example. However it has a new perspective in handling the GUI elements and events. I want to deviate from the Java SE as little as possible.
Here comes **SwingWT** into the picture. SwingWT is a free open source java GUI framework which operates over the SWT library providing the Swing API. It allows us to use the SWT library through the Swing API. For this they implemented the Swing API (or the major part of it), and these classes delegate calls and events to the SWT layer back and forth. A ready Swing application can be converted to use SwingWT simply by replacing the *java.awt* and *javax.swing* package names with *swingwt.awt* and *swingwt.swing*.

4.2 Software components and tabs

BWHF Agent provides functionalities and features for different areas related to Starcraft. These functions have to be separated on the GUI so the user can easily understand it and not confuse the features.
The different features are grouped and displayed in tabs. The common attributes and behaviour of tabs are implemented with the following model:



The *Tab* abstract class is the basis of all tabs. It defines the very basic features of tabs, and specifies 2 implementation-related methods:

- *assignUsedProperties()* is called when the tab must save all persistent properties it displays to the user for modification.
- *initializationEnded()* is called when the program has started and the main window is visible. Some feature might require this for proper layout (component sizes are not known until the window is displayed).

The *LoggedTab* abstract class provides logging features. If a logged tab has messages for the user, it can display them by calling the *logMessage()* methods. The class will display them in a consistent way, and it also logs messages to files. The logged tab also adds some log control GUI elements automatically such as the message text area scroll lock and clear buttons.

The *ProgressLoggedTab* extends the logged tab's features with a progress bar: when a descendant class has time consuming work to do, it can use this progress bar to display its current status.

4.2.1 Client properties and settings

BWHF Agent operates based on numerous properties and settings that can be changed by the user. These properties are stored in a java *Properties* object. When the agent is started, it builds a properties object holding the default values. After that it tries to load the stored properties. If the

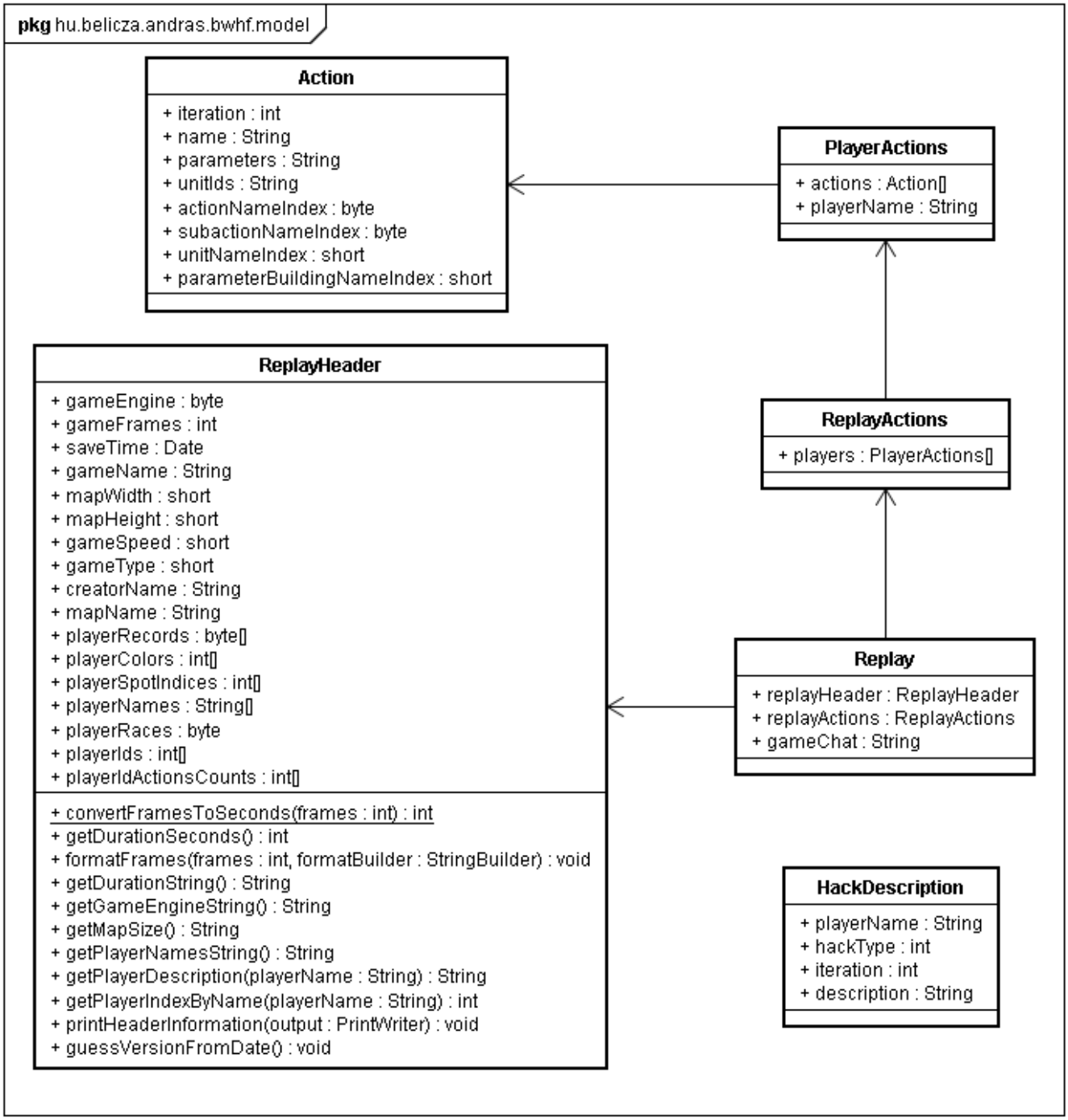
load fails (no previously saved properties for example) or the properties file is for previous version, all missing properties will fall back to the default properties silently without raising an error.
On startup the saved (or default) properties are loaded in the UI elements that display their current values and allows the user to edit/change them. On agent shutdown the *assingUsedProperties()* will take care of storing the current settings to the properties object.
Persisting the settings are simple done using the *Properties.load()* and *Properties.store()* methods.

4.3 MVC architecture

Examining the agent's architecture from the point of being a replay scanner program we can point at 3 well-defined parts of the code. These 3 parts realize the *model*, *view* and the *controller* components of the client's MVC architecture. This architecture gives us the possibility to use the hack scanner engine in other applications, or just replace the view layer to initiate scans and display results differently.

4.3.1 Model

The model is located in the *hu.belicza.andras.bwhf.model* package. This is the model's UML class diagram:

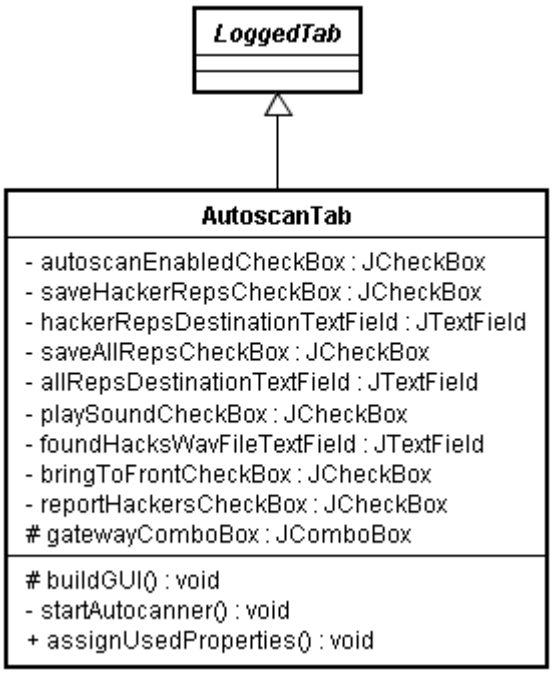


The model describes the Starcraft replay and its parts which are needed for the hack detection. The basic unit is the *Action*. This represents a single action given by a player. The actions are grouped by players because there are patterns which include multiple actions given in a sequence, and actions coming from different players has no meaning. The class *HackDescription* models the result of the hack scanning, more specifically it describes one detected hack event (which may involve multiple actions in case of a pattern).

4.3.2 View

The task of the view part of the replay scanner program is to provide GUI control elements to select the source of the scan, start and control the scan and display the scan results. The view part related to the hack scanner program is implemented as 2 tabs: one for the autoscan, and one for the manual scan.

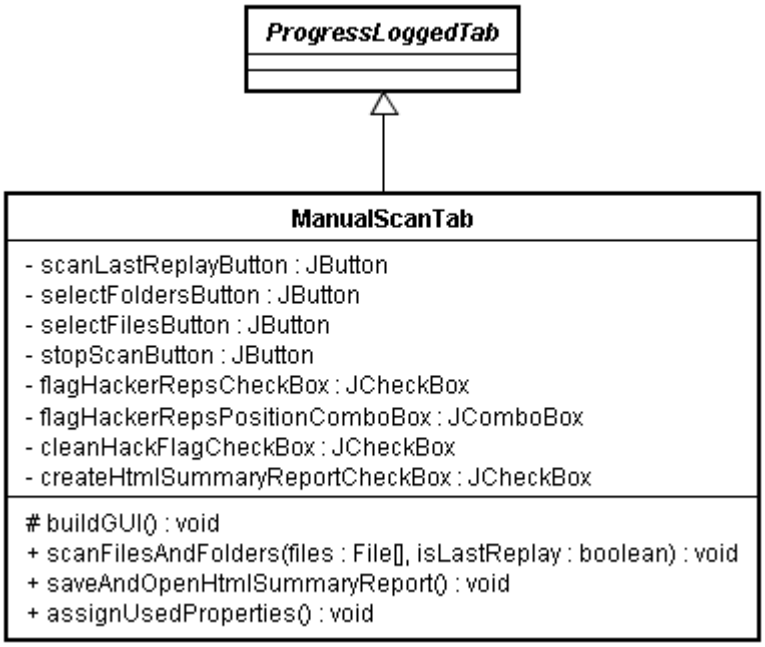
The autoscan tab provides GUI elements to control the autoscan and to display its status and messages.



Since the autoscan detects changes in the LastReplay.rep file, it very easy to implement another useful feature here: *replay autosave* (archives the lastreplay.rep when it changes). Replays where hack is detected can be separated to a different folder. The periodic check, the scan and the report runs in its own thread. The results are simply forwarded to and displayed in the logged tab's text area.

Autoscan runs automatically without user intervention. The source of this scan is well defined: it is always the LastReplay.rep file. There is no need for control buttons for autoscan.

This is not the case of manual scan.



Manual scans are triggered by the user. We provide 3 buttons to select the source of scan:

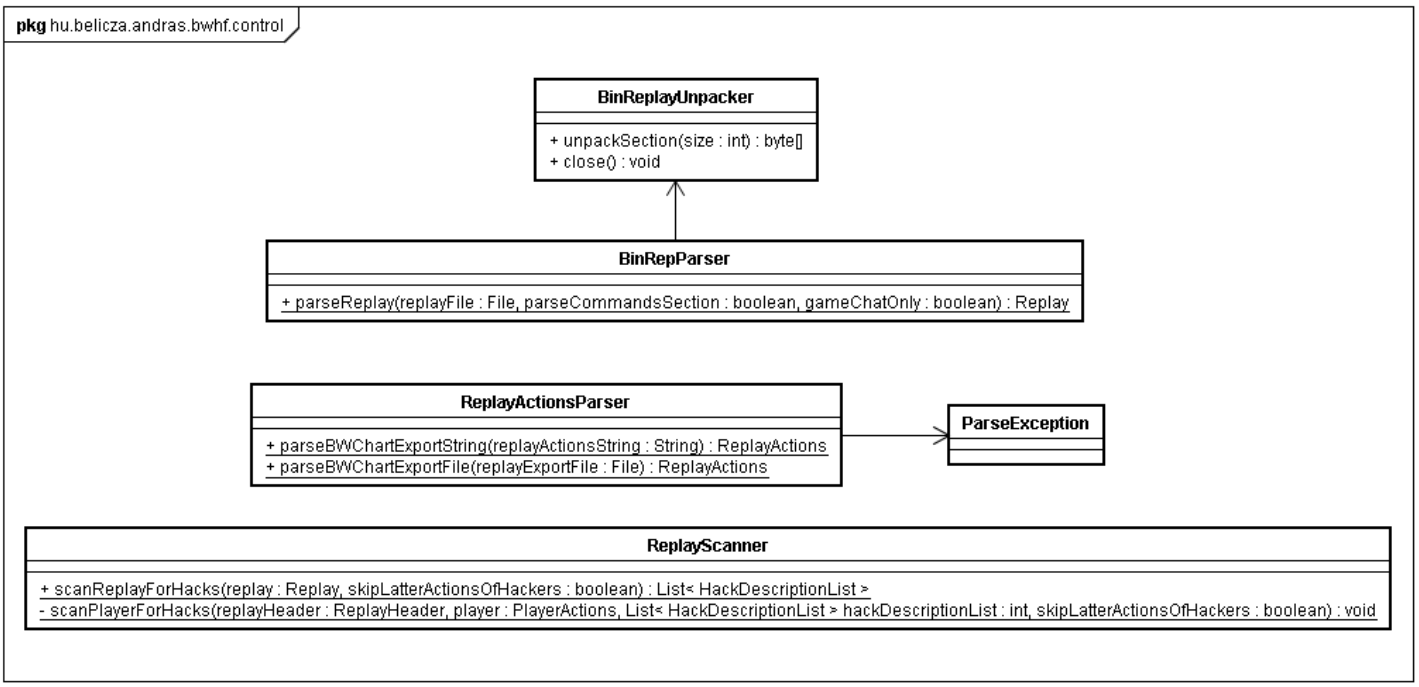
- *scanLastReplayButton*: it is a very frequent action to scan last replay after a game (mainly used if autoscan is not enabled)
- *selectFoldersButton*: opens a folder selection dialog, where the replays in the selected folders will be scanned recursively
- *selectFilesButton*: we can select explicitly the replay files we want to scan

When the source of scan is selected, a new thread will be started for the scan. That way we can update the GUI on the fly, and monitor the scan process. We can interrupt the scan at any time. Some useful features here include flagging the replays where hack was found by renaming them. We can choose to append a *'hack'* flag either to the beginning or to the end of the file names. The progress of the scan in continuously indicated in the logged tab's text area and on the progress bar. At the end of scan, a simple summary is written to the log text area enumerating the names of hackers.

Moreover a detailed HTML summary report can be created and opened in a browser. This detailed report contains the details of the scan (like when it happened, number of replays scanned, number of replays where hack was found), the list of hackers, the replays they hacked in and a description of hacks they used.

4.3.3 Controller

The controller is located in the *hu.belicza.andras.bwhf.control* package. This is the controller's UML class diagram:



The controller's task is to produce a *Replay* object from a source of a replay and execute a scan on it.

The controller package implements reading from 2 different source of replays:

1. Text files created by exporting actions from the BWChart application.
2. The original, raw, binary replay files created by Starcraft when saving a replay.

Parsing actions from BWChart export files is done by the *ReplayActionsParser* class. Its 2 methods can parse actions either from a text file or the text content directly passed onto it. Since these texts only contain the actions, it returns a *ReplayActions* object.

The first method was implemented earlier in the history of BWHF, but after I implemented parsing from binary files, the first method lost importance. Could be important for others who might use this as a library. It was used mainly in the web scanner form of BWHF (where one could copy/paste the exported text of BWChart into a text area to scan the actions).

The second method takes an original, binary replay file created by Starcraft itself. It is implemented in the *BinRepParser* class. For different use of this class, the parser method can be parameterized to customize what to extract from replays: replay header is always included, and we can choose to extract game actions or just the in-game chat.

The Starcraft replays are binary compressed files. The compression algorithm is a variation of PKWare's DCL algorithm. The *BinReplayUnpacker* class is responsible to handle the decompression of the compressed data found inside the binary replay files. The decompressing codes were taken from another project called Bwreplib which is part of the BWChart, I just ported and optimized those to Java environment (the original code was written in C/C++). The algorithm mostly operates on raw bytes.

To use bit operations in Java, it is important to convert bytes and int back and forth. In Java this is a bit tricky since there is *no unsigned byte* just *signed byte*. To convert a byte to an int, we have to use *byte_value & 0xff* instead of *(int) byte_value* else the signum bit is shifted to the most significant bit (conversion does not change negative numbers in the range of -128..-1, but the location of signum bit differs). Converting an int to a byte works the usual way: *(byte) int_value*.

The replay files consist of so called *sections* which can be decompressed independently more or less, so we can choose for example to extract only the replay header or the actions too, or the map section.

4.3.3.1 The ReplayScanner class

The *ReplayScanner* class is the actual implementation of the hack scanning and hack recognition algorithms and logic. The goals of the hack scanner engine:

- Perform scan on the unit of a *replay*.
- There are cases when we just want to know *who* and *if* he hacked in a replay, we do not care how and how many times. This *mode* is sufficient in most of the cases, to the average Starcraft user. This is based on *trust*, that they *believe* in the *author* of the program (which we can say).
- There are cases when we want to know exactly *when* and *how* someone hacked in a replay. This *mode* is important to advanced Starcraft players, admins and tournament organizers because they usually have to prove how someone hacked.
- Since scan may be performed on hundreds or thousands of replays, it *should* be fast.

The entry point of the replay scan is the *scanReplayForHacks()* method. It takes a *Replay* object as an argument to be scanned. The mode is specified by another argument, a shortcut flag: *skipLatterActionsOfHackers*. If this flag is set to *true*, then if a player is found hacking during a replay scan, his/her scan ends immediately. This shortcut speeds up the scan in cases where the hacker used his hack several times in a replay, when he used hacks early in the game or when the game went on for a long time after he used a detectable hack.

In harmony with the *model*, the scanner carries out the scans to one player at a time. Actions of players are the basic unit the scan engine operates on. Actions of a player has no effect on actions of other players in regard to hacking.
The *scanPlayerForHacks()* method contains the hack recognition code. It is responsible to scan the actions of 1 player.

The hack detection starts with a check for autogather/autotrain. This hack leaves an easily recognizable pattern in the beginning of the replay. The autogather/autotrain hack directs all starter units to gather from different mineral patches and automatically trains a new worker. All these actions are given in such early and in such a small time that is impossible by a human. Autogather/ autotrain is used by many and it can be recognized in an instant.

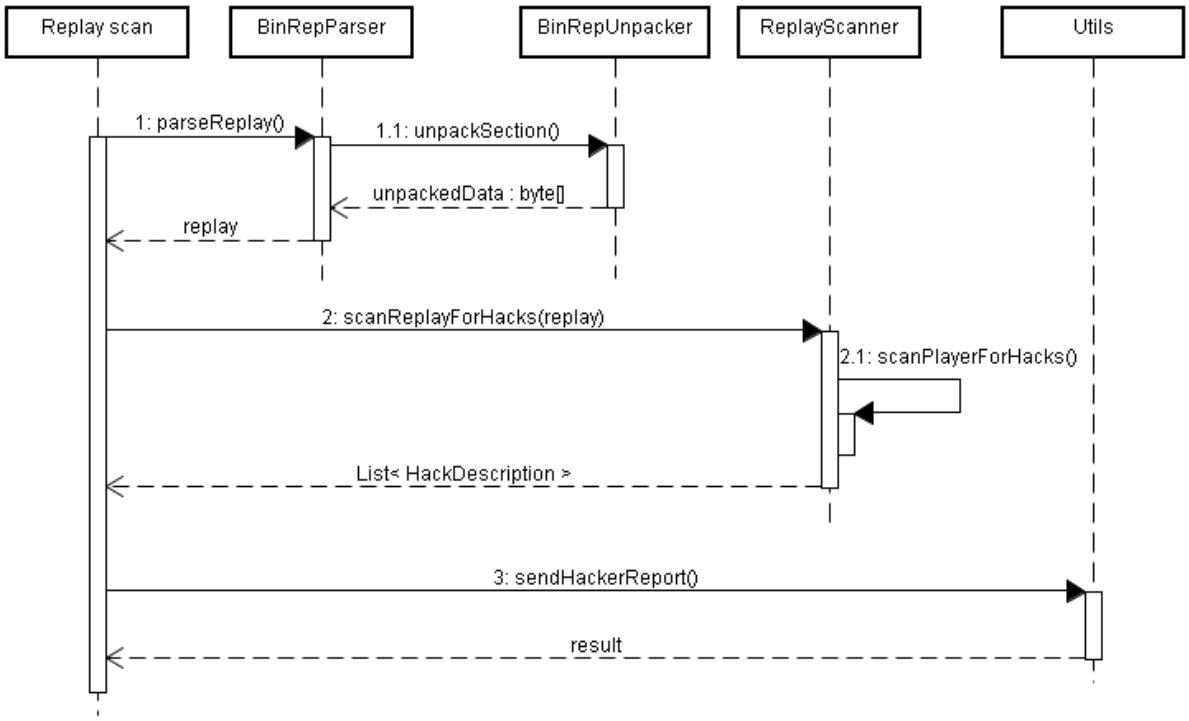
Next we iterate through all the actions of the players scanning for other hacks in no other particular order.

Currently the following hacks are detected and recognized by the scan engine:

- Use cheat drophack: if we find a *use cheat* action in a multiplayer game, it is proof of hack. This action is allowed in single player mode, but is not possible in multiplayer mode. Giving this command usually results in players being dropped.
- Ally-vision drophack: Ally-vision commands are used to set who a player is allied to and who he gives vision. In non-UMS type games a player always has to ally himself. If such ally-vision commands are given which does not set alliance to the sender himself, it is proof of hacking. This is also used to drop players.
- Build anywhere hack: If we find a build action which aims to build a building to a location where the building would reach out from the map, it is proof of hacking. Starcraft itself does not allow giving such commands. We know the sizes of the buildings, and the map size is recorded in the replay. This is easy to detect.
- Building selection hack: Only selection of 1 building is legit. If a select action has multiple building parameters, it is a hack. We have to be careful here, because in case of the Zerg race drones can morph into buildings. This detection cannot be applied to zergs.
- Moneyhacks: these hacks use well-known invalid actions. If we detect one of them, it is proof of hack.
- Multicommand unit control hack and multicommand rally set hack: these 2 types of hack result in the same pattern in the replay. The multicommand unit control hack controls more than 12 units as one group mapping them to smaller groups. The implementation of this hack repeats the commands given to the *big* group to all the small groups with the same parameters of course. The repeated commands are given at the same iteration. Selecting the different groups and giving the same commands to the same target in 0 time is not possible manually without using *hotkey* commands. This is proof of hack. The multicommand rally set hack does the same: the hack user virtually selects multiple buildings and sets the rally point. The hack after that repeats the *select building* and *set rally* command pairs to each virtually selected buildings. The hack does this in the same iteration.
- Multicommand hack: the *general* multicommand hack refers to giving several commands with a precision or speed that is not possible by a human. Care must be taken when we

decide *what* and *how many* actions at the same time are considered multicommand hack because when game lags, actions might spike up around the time of lag, causing seemingly impossible actions count at some frame. If actions being next to each other are the *same* actions (regardless to their parameters), it can easily be due to lag and/or *„action spam"* (when a player holds down a key for example). We must not count and report those. I also do not report *hotkey* actions since they can be spammed at will with the keyboard. A line must be drawn. Taking lag into consideration, examining thousands of replays, and the program being tested on tens of thousands or hundreds of thousands of replays, I can safely announce this: *If more than 20 non-hotkey, non-ally, non-vision, different actions are given at the same iteration, it can be looked at as proof of hack.*

Now we have an overview of the replay scan process:



4.4 Other useful features

In order the agent to work, we have to run it while we play Starcraft. In order to scan replays for hacks, we had to implement the replay format. Now that we have these, it is very easy and very useful for Starcraft players to implement other features. These features does not relate directly to hack recognition, but in many cases they make it easy to understand the recognition of hack, to easily find more replays where hacks might have been used..

4.4.1 Charts

The charts tab takes a replay as its input, and draws different types of charts based on the information being in the replay. The charts help to analyze the players and study their game play and certain hack scenarios.

Different types of charts of players which helps to analyze players and study certain hack scenarios. We can see the list of the players' actions below the charts, so we can study when, how and what were the players doing during the game. This list can be synchronized to the charts: if we navigate in the list or we click on the chart, we can see the actions that happened around the marker we just selected. Any part of the action list can exported simply by selecting and copying it to the clipboard.

We have a built-in search for this action list, and we can filter actions. This feature filters out actions that do not contain the entered filter text. It can be used for example to filter down to any select actions (like Select, Shift Select, Shift Deselect, Hotkey Select etc.) or to find any actions that relates to the unit Zealot (Train Zealot, Upgrade Zealot speed). The filter text might contain several words which will be in logical AND connection by default. Writing out AND is not needed but it is not a syntax error. The filter terms *"train zealot"* and *"train and zealot"* are equal. However we can use logical OR connection too by explicitly writing *or* between words. These 2 logical operators (AND and OR) can be combined in any way. The logical AND has a higher precedence than OR.

4.4.1.1 ChartsComponent class

The charts are located in the *ChartsTab*. It contains control elements for selecting the source replay of the charts. The *ChartsComponent* is responsible to display the chart's control elements and to paint the chart.

The ChartsComponent displays the following well separated elements:

- **General chart settings:** First we have a combo box to select what kind of chart we want to see. And we have the common settings here that apply to all charts. For example whether we want to display all players in one chart, or to have one for each. We can select to use players' in-game colors for charts, this helps to easily associate a chart with the player in the game. It is useful to disable players who did not really play just observed the game: a checkbox for disabling players having relatively few actions.
- **Selected chart's settings:** Each chart has its own characteristics and settings. Those are grouped to this box.
- **Players:** We see a list of checkboxes for each player. We can enable/disable any player we want independently.
- **The chart canvas:** the picture of the charts. The graphs will be drawn here.

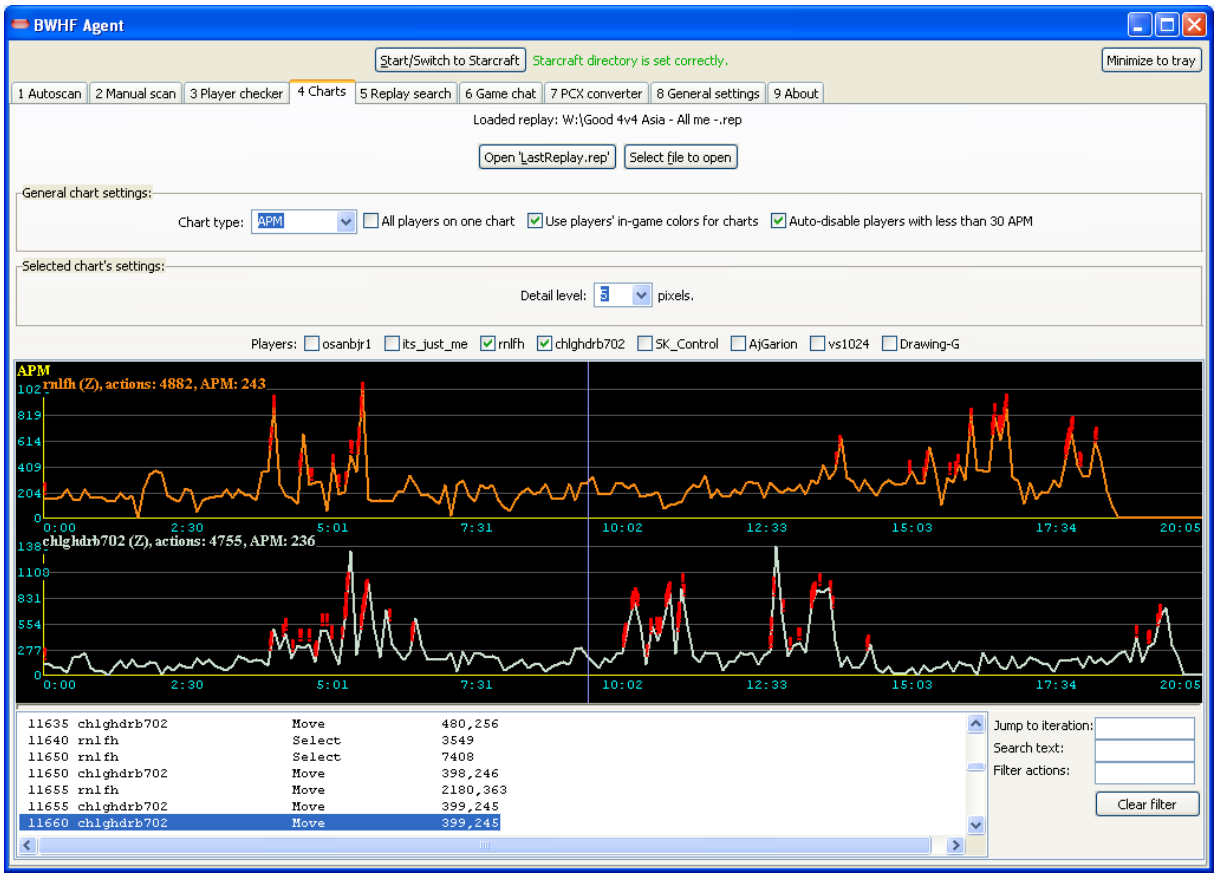
- **Player action list:** a component listing the actions of the game.

The full list of actions of the game is stored in an array. Every time when a filter operation is given, we recalculate the visible part of the list. Since one game might contain tens of thousands of actions, a simple *JList* would become very slow. Instead I use a simple *JTextArea* to display the actions. This is much faster. To give some kind of list feeling to it, when the text area receives actions (such as mouse clicks or cursor mover keys), an automatic line selection is executed. To synchronize the action list to the chart marker, I used the binary search algorithm in the actions.

The ChartsComponent is responsible to draw the different charts. There are common tasks for each of the charts, for example drawing the axis, labels of the players etc.
Each of the chart is drawn to fit the available space. This means using the width of the window, but the height might vary because the canvas and the action list is divided with a *JSplitPane*. The sizes of graphs even vary due to the setting of having charts for each separate player or putting all to one chart.
The logic of scaling charts to always use the available space based on all the conditions is implemented in the *ChartsParams* class. It calculates the sizes (width and height) of the charts, and has helper methods to compute the x and y coordinates of the charts for the different players. Based on the length of the game (number of iterations) it can convert back and forth x coordinates to iteration. This is used for the synchronization of the marker and the action list.

4.4.1.2 APM chart and Overall APM chart

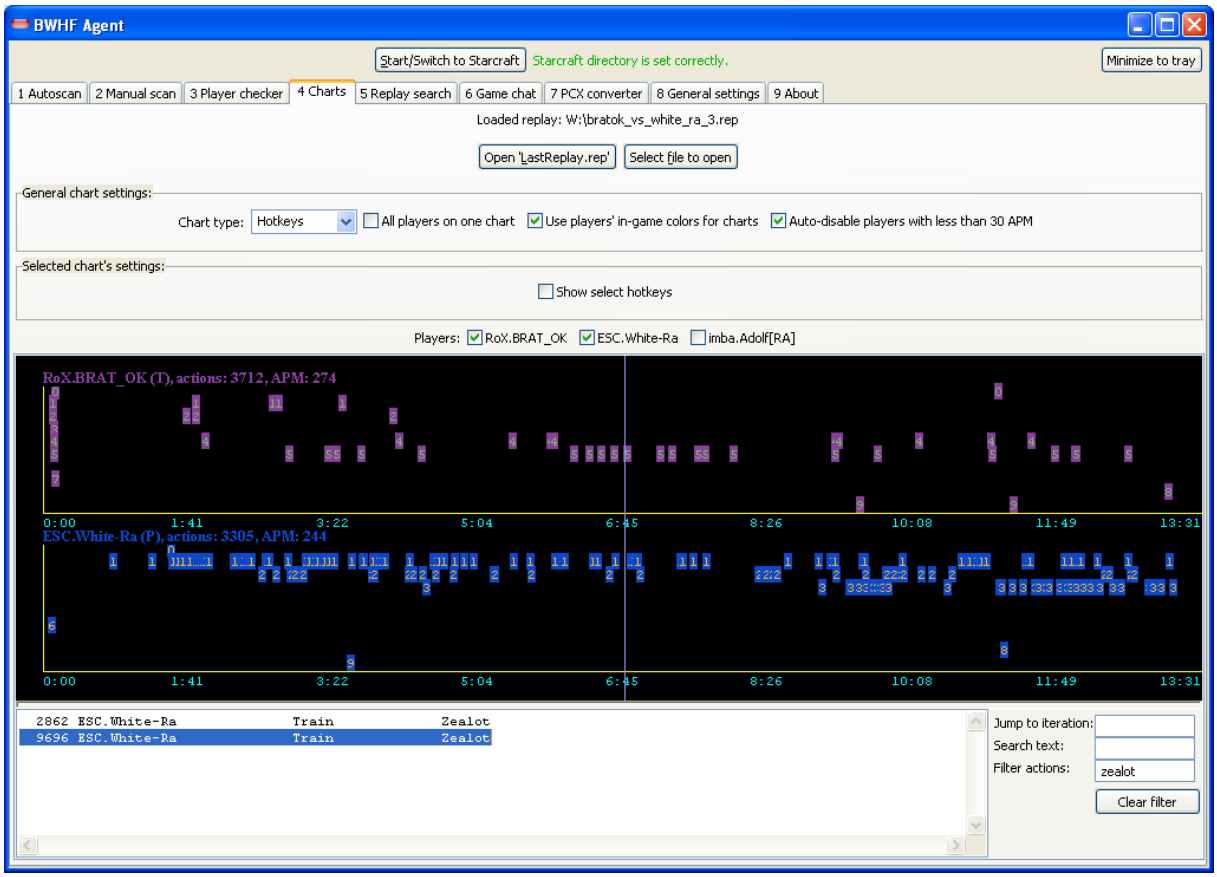
The **APM** chart is the most commonly used chart. It displays the players' **action per minute** rate at every moment. Every hack event is marked on the chat with an exclamation mark. The hack events are of course coming from a full analysis by the *ReplayScanner* class.
Usually when general multicommand hack or multicommand unit control hack is used, it results in spikes in the APM charts. We can see this in the charts below.
The **Overall APM** chart visualizes the APM calculated for the whole game from the beginning at every moment).
There is one setting for these types of charts: the *detail level* in pixels.
The APM charts in my implementation are in fact *polylines*. The detail level will tell the granularity of these polylines. The neighbors of the points of this graph will have a distance of the detail level. The graph value will be the actions during this interval scaled properly.
The usefulness the APM charts regarding to hack detection is not even questionable.



4.4.1.3 Hotkeys chart

Next we have the **Hotkeys** chart. This chart visualizes the players' hotkey usage. The players can assign a group of unit of a building to a number, and that group or building can be recalled anytime that number is pressed. Every decent player uses hotkeys. If we see a player playing relatively well but has zero hotkeys, the player falls under suspicion immediately.

The hotkeys chart has one setting: we can enable/disable the select hotkeys (by default only the assign hotkeys are visualized).

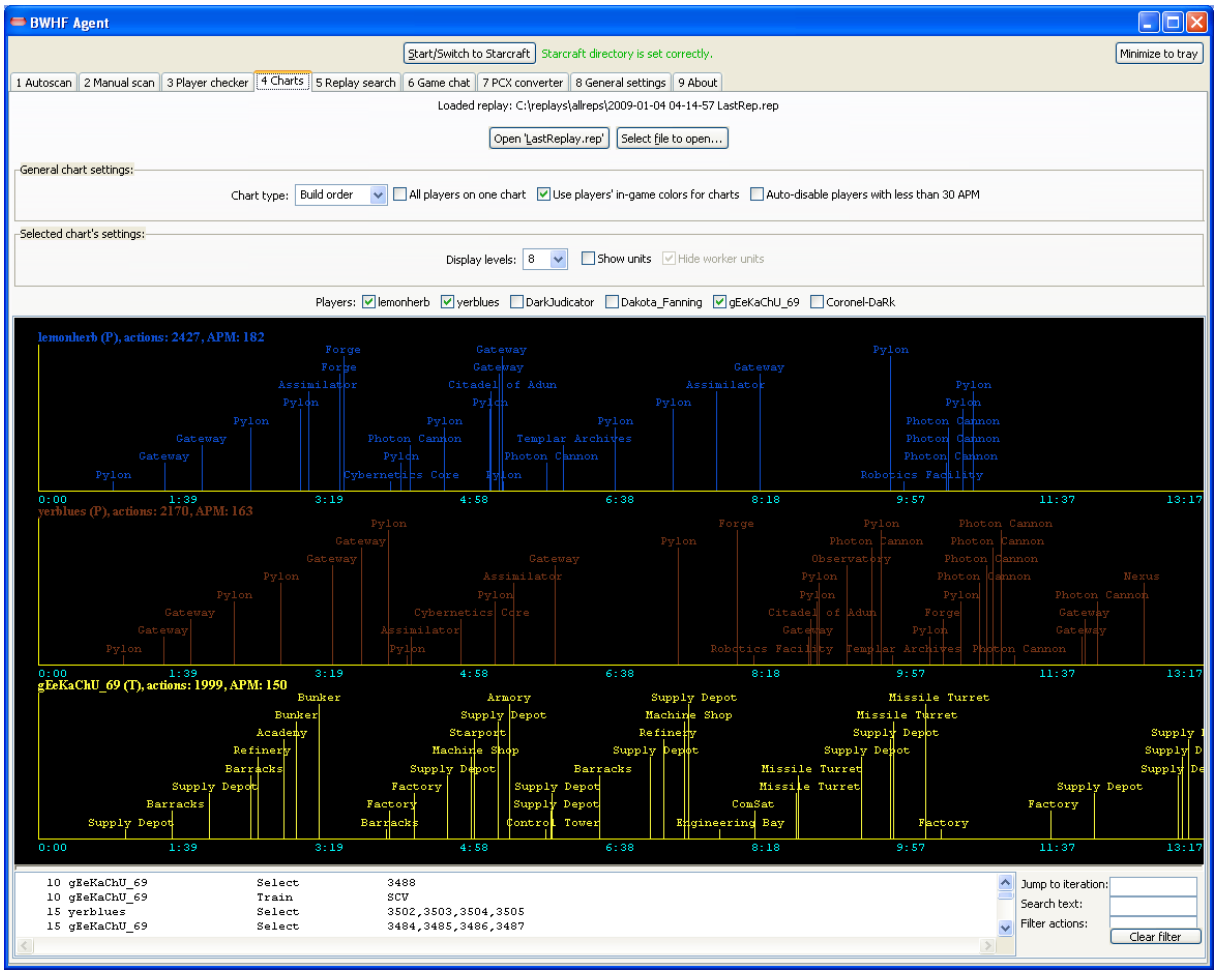


4.4.1.4 Build orders and Strategy charts

The **Build orders** chart displays the build commands in a time line. Since there are relatively many build commands, it displays the build actions varying in *depth* or in the *y coordinate*. This can be customized with the *display levels* setting. The *level* has no particular meaning, it just helps making the chart more readable.

When money hack is used early in the game, the extra money is often used to construct buildings which would not be possible at that time. Experienced players can easily tell about such early builds that they are hack.

The **Strategy** chart is very similar to the build orders chart: it displays events in a time line using different levels. The difference is that the Strategy chart displays different strategy actions and decisions as events such as building defense, expanding, dropping etc.



4.4.2 Game chat extractor

In-game chat is recorded in the replay. This tab gives the possibility to extract this information. The Game chat tab is a *ProgressLoggedTab*.

Any replay files can be selected, and we can decide whether just display the text or to write them to text files. Along with the game chat useful information can be extracted from replays such as players, duration, map name and size etc. Having the binary replay extractor, this task is nowhere near difficult, but is very useful for Starcraft players.

4.4.3 PCX screenshot converter and auto-converter

It is possible to create a screenshot of the game at any time by pressing the Print screen key. However, Starcraft is a very old game, and when they implemented this feature, they did not really guess what format would be suitable or useful for the players. Probably due to its easily implementable format they chose the PCX format. Today this format is almost became extinct. The professional picture and photo editors know it of course, but the single user *does not know* what to do with it and *does not want* to be bothered installing editor or converter programs.

Converting PCX screenshot files to well known and supported formats (such as GIF, PNG, JPG, BMP) are a very handy feature of the program.

For the PCX format I downloaded and used a Java PCX ImageIO service provider implementation from the internet. The program lists the output image types supported by *ImageIO*, the conversion is simply an *ImageIO.read()* and *ImageIO.write()* call pair.

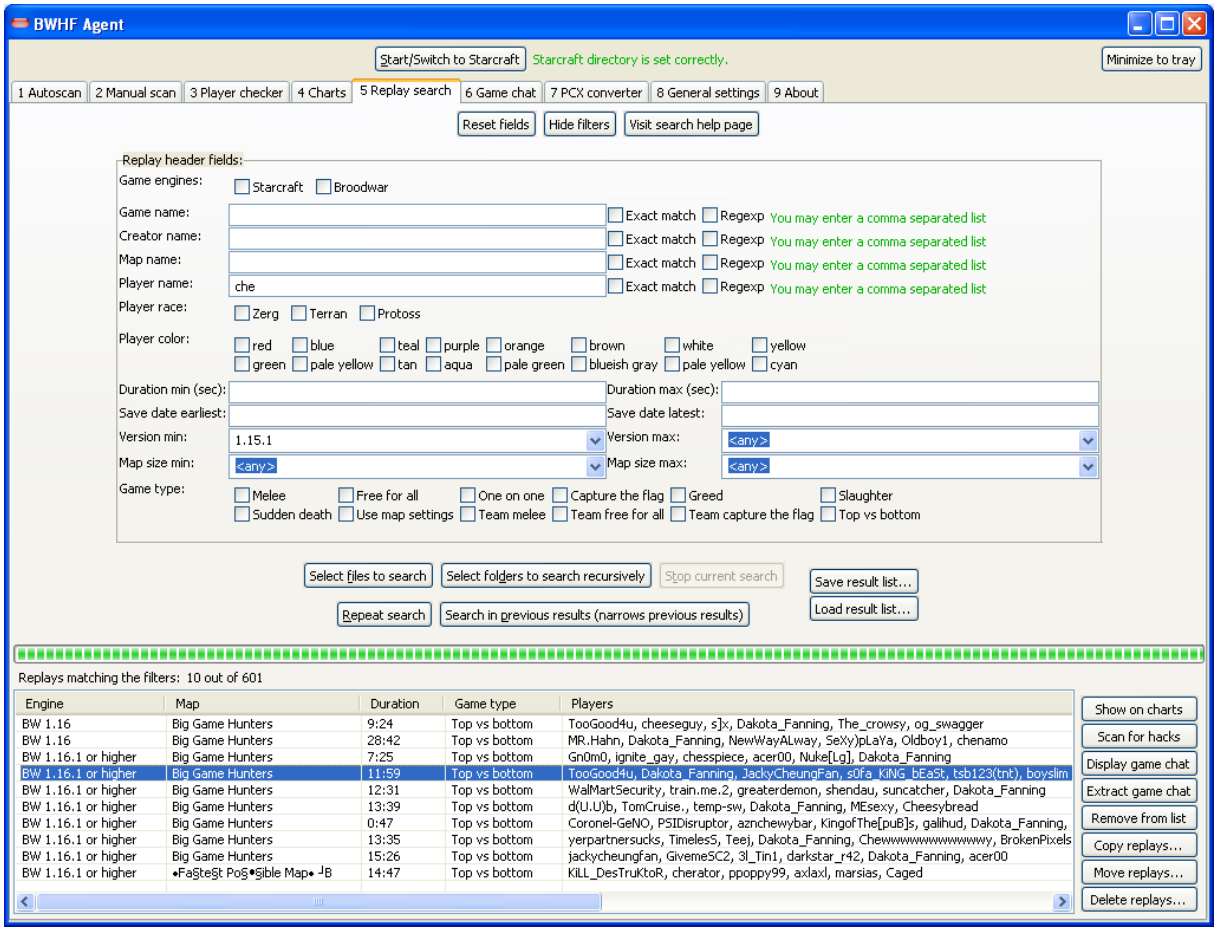
The PCX converter tab is a *ProgressLoggedTab* too.

4.4.4 Replay search

Replay search is a very basic operation for *"veteran"* players who have hundreds or thousands of replays. It is necessary to easily find any replay matching different conditions.

Regarding to hack detection, if we found a player hacking, we can search for all the replays of the same player to find more hacks or hacker replays. Sometimes people change names, so it might be better some cases to search for the same game name or replays saved within a small time frame.

BWHF Agent has a very powerful search tab which looks like this:



Goals of the replay search were:

- Provide a search function for beginners who just simply want to find a replay based on 1-2 simple condition.

- Provide a comprehensive, sophisticated replay search for advanced players or even advanced IT users who can take advantage on tools such as *regular expressions*.
- It should be as fast as possible, no one likes to wait, especially those who use search often.
- It would be nice to be able to save and load search results.

To meet the goal of being easy to use: if we don't touch any of the search fields, the search returns all the input replays. If we want to search just for one condition (like a player name), we enter the player name and execute the search. If we specify more conditions, they will be in logical AND connection.

4.4.4.1 Replay filter fields

There are different types of filter fields. When there are some predefined values, we can see them listed with checkboxes. We can check any that we are interested in. They will be in logical OR connection: if any of the selected values apply to a replay, the replay will be included in the results.

Values entered to text fields are case insensitive and will be trimmed: redundant spaces will be removed from the beginning and from the end of the entered values. Text fields have an *Exact match* option: if this is selected, the entered value must match exactly the replay property they apply to, else the entered value may be a substring. If exact match is not used, we can enter either a single value, or a comma separated value list. Text fields also have a *Regexp* option: if this is checked, the entered value will be interpreted as a *regular expression*.

There are some fields which can have values from a wide range. In these cases the filter is an interval filter: we can define the minimal and the maximal accepted values. We are allowed to only set one of them, and then the other end of the interval will not be limited. When the interval is limited to some valid values, we can see a combo box with the valid values, and we can select the one we wish to set as the limit.

4.4.4.2 Search execution buttons

Search can be executed by specifying the set of replays to search. The currently running search can be terminated at any time with the **Stop current search** button.
There are several buttons to define the source replays of search:

- **Select files** button: we can select multiple replay files to search.
- **Select folders to search recursively** button: we can select folders of replays to search recursively.

- **Repeat search** button: the search will be repeated on the same replays as the last search but with the newly specified or modified filters of course.
- **Search in previous results (narrows previous result)** button: the search will be executed on the result of the last search. We can filter previous search result with this button.

Filter fields having syntax error will be excluded from the search and a red background will notify the user.

4.4.4.3 Search results

The number of searched replays and the number of replays matching the filters are displayed above the result table. The result table lists every replay matching the filters. We can select any of the replays, and we can perform various operations on them with the buttons being on the right of the table. If we double click on a replay, it will be shown in the Charts tab.

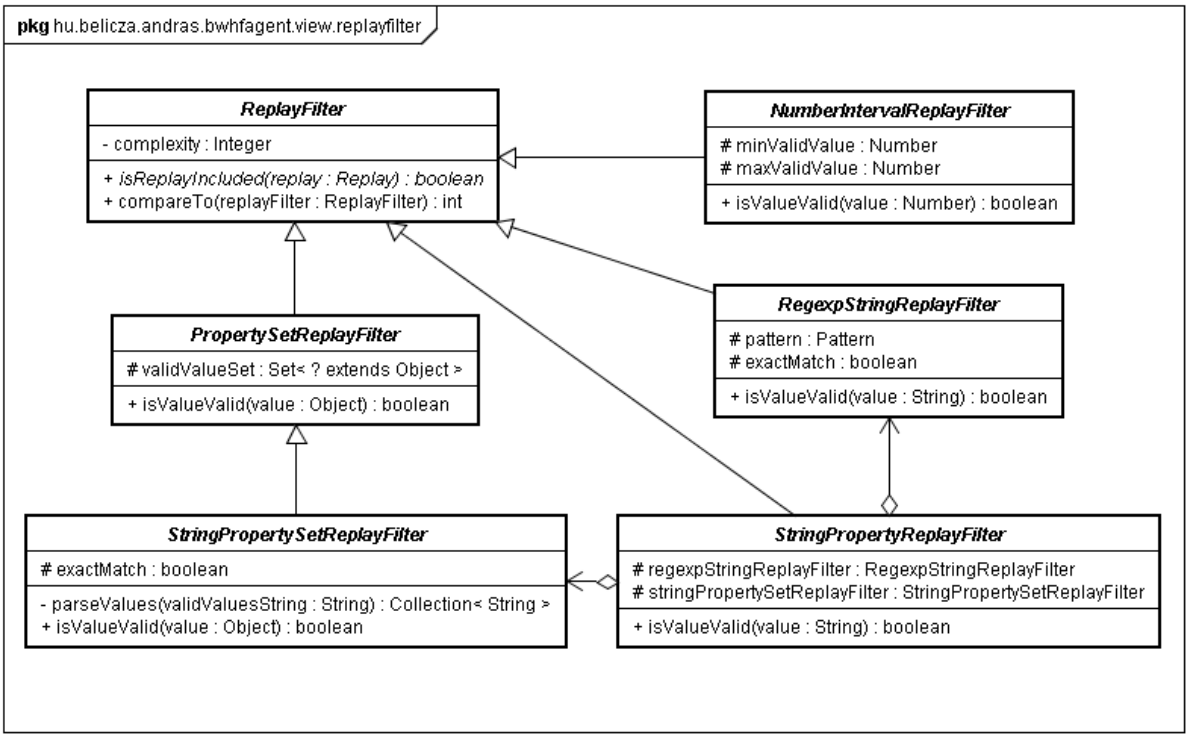
Search performance

I tested the search on 10 thousands replays with an average computer (3 GHz CPU, Windows XP). The search executed in like 1 minute, and the result table was constructed in another like 20 seconds. I repeated the search, and it completed in like 3 seconds.

4.4.4.4 Implementation of the replay search

I model the different types of search fields with different classes. For the efficiency I define the complexity of the search filter: a relative number which tells the computing capacity required to check if a filter applies to a replay. When the user enters the search terms, we create filter objects representing the terms, and sort them by complexity. Since all filter has to apply to a replay in order to put it in the results, we can check the faster ones first.

Here is the UML class diagram of the different types of filters:



The *ReplayFilter* class stores the complexity and defines an order based on that. In the subtypes of ReplayFilter the complexity is known based on the concrete type of filter. For speed and efficiently, all filter objects and all of their fields are computed before we start to apply the filters to replays.

The concrete filters which applies to the concrete properties of replays are direct subclasses of the presented abstract filter classes. For example the filter field for the player name is called *PlayerNameReplayFilter* and it is a direct subclass of *StringPropertyReplayFilter*. It can handle both regular expressions and plain texts, and can match whole words or substrings in both cases.

Since the filter properties are all stored in the header section of the replays, we only extract the headers of the replays which is relatively very small (279 bytes) compared to the common size of replays (30KB-200KB). This is the strongest factor in being a lot faster than other replay search tools and programs.

The result list is a table. When saving and loading the result list, the *CSV* format is used to store the values (but the *tab* character is used to separate values). This way the exported list can be opened with Excel for example. When we load a result file, we handle the order of the columns (means the result list and the result table can have columns in different order). Missing columns are left empty, extra columns stored in the file are discarded.

4.4.5 Player checker

People do not like to play with hackers. Unfortunately BWHF Agent only tells this after the game (this is the price of being legit). Even if we would know it during game, not much we could do.

However, if we have a big list of hackers (which we do thanks to the wide usage of BWHF Agent), we could check players before game starts whether they have been reported previously to the BWHF hacker database. Switching to the web interface of the database and manually typing the names of players is very unpleasant and unlikely anyone would do. It *would be best* if this check would be automatic.
Checking players before game and alerting the user if hackers are in the lobby is the most anticipated feature of BWHF Agent.

There is a big problem though: if we want the program to be legit, we cannot read Starcraft's memory. So the question remains: how can we obtain *automatically* the players names without reading Starcraft's memory? By *automatically* I mean the user does not have to type the names.

There is one solution, my brand new idea (I have not seen this technique used in any games or game utility programs):
In the game lobby if we press the "*Print Screen"* (or *"PrtScn"*) key in order to indicate that we want the players to be checked, Starcraft will save the screenshot to a PCX file. This new PCX file is what BWHF Agent is listening to. If new PCX screenshot is detected, BWHF Agent processes the image and uses a text recognition algorithm to read the player names from the image. After that it can check the player names in the BWHF hacker database.

The *PlayerCheckerTab* works in cooperation with the *PcxConverterTab*. New PCX file detection is implemented in the PcxConverterTab. The PcxConverterTab calls the PlayerCheckerTab with the new PCX files, and the PlayerCheckerTab can process them first.
The PlayerCheckerTab first checks if a file is a game lobby screenshot. If yes, processes it and optionally deletes it. The remaining (not deleted) PCX files are returned to the PcxConverterTab for further processing (conversion).
Deciding if a screenshot is a game lobby screenshot is very simple, we just have to check certain parts of the image (I simple just test if a red line is present in a specific location).

If every time we would connect to the central hacker database when a BWHF Agent user presses the Print screen key anywhere in world, that might would overload the server. And moreover that would slow down the player check process. Instead of this we store a local cache of the hacker list, the fewest data that is required for check: the gateway and the name of the hacker. This local cache can be updated at an interval of the user's choice or at will at any time. The check only applies to the hackers having the same gateway as set in the autoscan tab. The users can even define a custom list of player names (and gateways), and they will be checked too.

If a match is found in the local hacker list cache, a voice *"hacker at slot x"* is played, so the user does not have to switch to BWHF Agent to know who the hacker is. If someone from the custom list is detected, then the voice *"custom at slot x"* is played.

4.4.5.1 Text recognition

This is a simple case of text recognition because the recognizable text is drawn using 1 font (1 style, 1 size, text properties such as italic or bold do not change) and player names are written at well-defined positions. The text color may vary though because brightness inside Starcraft can be changed.

I created and saved a *chardef* image which contains the images of all characters we are interested in. When the agent starts, loads this image and parses the characters: characters and their order is known, and they are separated with an empty column. The images of the characters are built using a few different colors (4). I model the images of the characters with a matrix storing the color index of the pixel at the position specified by the indices of the element.

Later when we have to recognize the names of players in the screenshots, we check if the image of a character is present at a location.

The player names can be at well-defined locations called *slot*s. It can be easily detected if a slot is present: it has a red frame. If we detected a slot, we try to recognize a string inside the slot. To speed up the process we first determine the width of the character by scanning for an empty line, and we only try the characters having the detected width.

Since the brightness in Starcraft can be changed, the pixel colors may vary. Therefore we cannot test equality with predefined palette colors. Therefore the character test is done in the HSB color model. Java has built in function to convert RGB values to HSB values. In the HSB model the H (hue) component tells us the *color* of the pixel. If this matches with the sample pixel, we accept it.

There is one disadvantage of this solution. Starcraft uses such font in the game lobby in which the capital *i* and the low *L* has the same picture, they cannot be distinguished.

In cases when the players' names contain any of these 2 letters, and there are a hacker in the database whose name only differs in the questionable characters, the word *"possible"* is played before each alert (for example *"possible hacker at slot x"*).

In cases when the recognized name contains questionable characters, we have to create permutations of the name, and all permutations have to be checked in the database.

For example if the hacker's name is "*pillow"* and he writes his name as *"pIllow"*, that gives us 3 questionable characters. From the 2^{nd}, 3^{rd} and 4^{th} character we cannot tell if they are capital I or small L's. We have 3 questionable characters, each might have 2 possible values, that gives us $2^3=8$ permutations. On the battle.net player names can be 16 characters long. That would give us $2^{16}=65536$ possible names. Generating and checking all those names would take way too long.

Since this is a very rare and unique case, we will not handle this. We limit generating permutations up to 5 questionable characters. If the recognized name contains more questionable characters, we will take them as small L's (but saying the "possible" word when indicating of course).

5 Iterative releases and testing

Today's software products must comply to the users wishes and demands. To achieve this, we have to call in the users to the development process as early as we can and cooperate with them. It is lot more profitable to recognize and fix bugs in the early state. In large IT systems the 90% of the functionality and features are never or very rarely used. It is recommended to implement those features first that the users use and find useful.

BWHF Agent is built in this moral. The first public release came out on December 28, 2008, and since then the development progresses iteratively. I released a new version in every 2-3 weeks, posted it on the main Starcraft sites, and listened to users feedback. Users test the program on hundreds of thousands of replays, and they do it in ways the developers often do not think of. Users give feedback after each iterative release what they like, what they miss and bugs they find in the program.

The project's web page contains a complete issue and bug tracking system. Users can file bug reports and enhancement requests, replays and screenshots can be attached.

6 Past, present and future

6.1 Project history

Everything began with just me analyzing my replays in BWChart, looking for hack actions and patterns. Sometimes I started managing lists of hacker, but without such program like BWHF Agent, it could not keep up with the massive amount of hackers.

One day I decided to include all the experience and knowledge I built regarding to hack detection into a web scanner project. This project is still available here:

http://icza.fw.hu/hu.belicza.andras.bwhf.BwhfEntryPoint/BwhfEntryPoint.html

The web scanner contains an early version of the hack scan engine. It takes the BWChart exported actions of a replay as input, and performs a scan on it. The result is simply displayed in a text area, but there was no central hacker list. This was a big step for hack recognition for the public, but lacked the capability to store the results. And it was very unpleasant to use: first the replay had to be opened with BWChart, actions had to be exported, then copied into the web scanners input text area. A *„Howto use BWHF"* video on YouTube demonstrates this: http://www.youtube.com/watch?v=u1wLAYq31Gk

The web scanner was built using the Google Web Toolkit framework.

The BWHF Agent brought this hack detection to a whole new level: BWChart no longer required, the hack scan is completely automatic, the detected hackers can be automatically reported to a central hacker database, and players can be checked before game whether they have been reported earlier.

Beyond hack recognition, BWHF Agent is one of the best utilities for Starcraft players: with the replay search, charts and screenshot conversion it is an all-in one package.

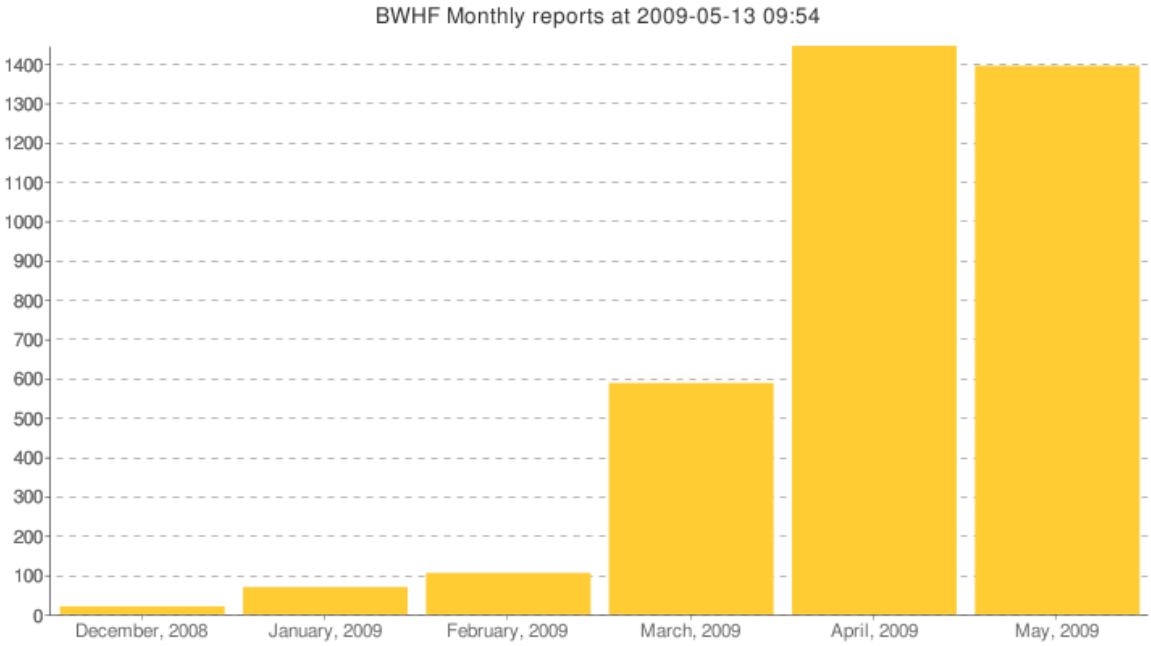
6.2 Program usage and database statistics

Downloads from the official web page are counted and can be viewed. The numbers shows that every new release is downloaded by hundreds of Starcraft players. I have received a lot of positive feedback. I received messages even from hackers that they stopped hacking and start using my program. I received messages from inactive players who told they stopped player due to the increasing and enormous hacking, but this project gave them a full reason to start playing again.

The web interface has a statistics page where we can view different statistics regarding to the reported hackers and the reports. We can see a chart of the gateway distribution between hackers, and the number of monthly reports.

The online statistics can be found here: http://94.199.240.39/hackerdb/hackers?op=sta

The number of monthly reports increases almost exponentially even though relatively few keys given out.



6.3 Project future

The project's main goal and feature is the hack scanning and recognition feature. We can continue to keep the hack scanner engine up-to-date: add the detection of new hacks and new variations of hacks.

Although not everyone is interested in this. On the other hand, every Starcraft player who play regularly needs and uses replay organizer and analyzer software. Since the replay format and processing is already implemented, we can continue to add replay organizer and analyzer features to the project, and then more people will use it. Of course if somebody uses the software for its organizer and analyzer features and he sees the detected hacks, eventually he might be interested in reporting them as well. This has happened before.

User interface can also be improved.

This Thesis completes with BWHF Agent version 2.21. The development did not end, a newer, 2.30 version has been released which focuses more on organizing replays, and improves the user interface by adding icons to tabs and buttons. User interface became much more friendly this way.

7 Bibliography

Nyékyné G. Judit (szerk.) et al.: Java 2 útikalauz programozóknak 1.3 I-II.

Nyékyné G. Judit (szerk.) et al.:J2EE útikalauz Java programozóknak

Rónyai Lajos - Ivanyos Gábor - Szabó Réka: Algoritmusok

Bócz Péter - Szász Péter: A világháló lehetőségei

Gamma, E. et al.: Design Patterns, Addison-Wesley, 1995

Bernd Oestereich, Time Weiliens: UML 2 Certification Guide

Java 6.0 SE: http://java.sun.com/javase/

HSQLDB: http://hsqldb.org/

SwingWT: http://swingwt.sourceforge.net/

Java PCX ImageIO service provider:
http://www.informit.com/content/images/art_friesen_pcx/elementLinks/code.zip

Bwreplib: http://bwchart.teamliquid.net/us/bwlib.php

Google Charts API: http://code.google.com/apis/chart/