# TribalWars Super-farm Bot 1.0

Main **purpose**: automate user farming scenario in TribalWars browser strategy game. This scenario is roughly the following:

1. User authorizes himself to server.
2. User opens Map game page in browser.
3. User a decision about which village to attack and how much Light Cavalry (LC) (s)he needs to send on this village. Decision making is usually boils down to the next questions:
4. Distance to the village (farm nearest villages first).
5. Population of the village (try to farm largest of nearest first)
6. Expected loot (based on expected resource h/rate production and time of last visit).
7. User sends an attack(s).
8. User waits for attack and then reads appropriate report.
9. User makes a decision, whether (s)he needs to send more LC to this village or select another target.

Basing on the aforementioned user-scenario, Bot’s actions could be described in the next steps:

1. Bot is started by main component
2. Bot authorizes himself to server and collects all needed cookies. (**Note:** This will not be implemented as part of initial version due to complexity of server authorization procedure).
3. Bot uses received cookies to build HTTP request templates (**Note:** there are 2 unique identifiers which should be in cookies: **cid & sid**, both will be passed manually by user)
4. Bot requests the Map page.
5. Bot parses the Map page and builds a mapping of all nearest Barbarian/Bonus villages which have no owner.
6. Bot requests village-Overview page.
7. Bot parses village-Overview page to get the number of LC which are in the villa.
8. Bot plans and sends attacks
9. Bot tracks attacks by time sent & coordinates.
10. When attack is made, Bot requests appropriate report page.
11. Bot parses report page and analyzes it.
12. Depending of report’s results, Bot either sends more troops to the same village or chooses another target.

## Classes

### Main

Main application flow.

**Responsibilities:**

1. Runs the **Bot.**
2. Stops the **Bot.**
3. Extract statistic from the **Bot** and passes it to some **StatisticManager** to build exhaustive farm report.

**Collaborators:**

1. Class [Bot](#_Bot).
2. Class StatisticManager.

### Bot

A component responsible for assembling and running all other components.

R-s:

1. Inits new **RequestManager**
2. Inits new **Map** with an instance of RequestManager.
3. Inits new **AttackQueue** with an instance of Map.
4. Inits new **AttackObserver.**
5. Inits new **TroopsManager**.
6. Gets a troops count from **TroopsManager**.
7. On start:

* Gets a next\_village from **AttackQueue**
* Inits new **Attack** with a next\_village (**Village**) and instance of self.
* Asks **Attack** if it is\_valid.
* If Yes: calls **Attack.send\_attack()** and registers time-values in **AttackObserver**.

If No: goes to constant time.sleep(1) until **self.needs\_refresh** flag will be set by an **AttackObserver**.

After that, asks **TroopsManager** for updated troops count and calls the same cycle recursively with the same **Village** instance & instance of self.

1. Asks **AttackQueue** to update **Map** with upon finishing farm & when crashes occur.

C-s:

1. class [RequestManager](#_RequestManager).

2. class [Map](#_Map).

3. class [AttackQueue](#_AttackQueue).

4. class [AttackObserver](#_AttackObserver).

5. class [TroopsManager](#_TroopsManager).

6. class [Attack](#_Attack).

7. class [Village](#_Village).

### Map

A collection of villages.

**Responsibilities:**

1. Calls **RequestManager** for a map overview (HTML) and builds the mapping of **Villages** in sector:

{(x, y): [**Village**, distance\_from\_base\_point], …}

**Notes:** Bonus villages which have owner (player) should be filtered out. Bonus should be passed to **Village** constructor (for correct h/rates)

1. Recursively calls **RequestManager** for a map overview from corner villages.
2. Stores the mapping of **Villages** in a local shelve file.
3. Responds to **AttackQueue** with a mapping of **Villages** filtered by a given distance limit.
4. Updates self from list of **Villages** in **AttackQueue**.
5. Methods:

* **\_\_init\_\_(**self, x, y, RequestManager, depth=1, mapfile=’map’):

self.RM = RM,

self.villages = {},

self.mapfile=mapfile

self.base\_x = x

self.base\_y = y

self.add\_saved\_villages()

self.build\_villages(x, y, depth)

* Self.**add\_saved\_villages**:

With shelve.open(self.mapfile): get shelve[“villages”]

If “villages”:

For each village:

Self.calculate\_distance(village) # from self.base\_x, base\_y

Self.villages[coords] = [village, distance]

* Self.**calculate\_distance**(x, y):

Use math.sqrt, return distance(int) of given (x, y) from base\_x & base\_y

* Self.**get\_map\_data**(html\_map):

Parse html\_map,

Convert from json to list

Return list containing map sectors

* Self.**is\_valid**(village\_data):

Checks village data for name:

Returns True if name == 0 or,

If name == “Bonus village” and owner == 0

* Self.get\_sector\_corners(sector\_coords):

Accepts list of sector\_coordinates.

Sorts a list 4 times by combination of key=’’ & reverse=’’ to get 4 points:

(x=min, y=min), (x=max, y=max), (x=min, y=max), (x=max, y=min).

Returns list of 4 corner points.

* Self.**create\_village**(coords, village\_data):

Builds village object.

If village name == ‘bonus village’, passes bonus to Village constructor.

* Self.**build\_villages**(x,y, depth):

Depth -= 1

Html\_map = Request s\_html\_map(x, y)

Sectors = Self.get\_map\_data(html\_map)

For sector in sectors:

X\_corner = sector[“x”]

Y\_corner = sector[“y”]

sector\_coords = []

For index, row in enumerate(sector[“villages”]):

For village in row:

Self.is\_valid(village\_data) # check if Barbarian or Bonus w/o owner.

Build village coords (x + index, y+y\_corner)

sector\_coords.append(coords)

If villa\_coords not in self.villages:

Self.create\_village(village)

Self.calculate\_distance(villa\_coords)

Self.villages[coords] = [village\_obj, disatnce]

If depth:

Self.get\_sector\_corners(sector\_coords)

For each corner in sector\_corners:

Call self.build\_villages(corner, depth)

* Self.**get\_villages\_in\_range**(distance):

Returns mapping of villages that lies in range=distance

* Self.**save\_villages**(mapping):

With shelve.open(self.mapfile):

If not “villages” in shelve:

Shelve[“villages”] = {}

For each village in mapping:

Insert (x,y): village\_obj into shelve[“villages”] # don’t need to store distance, #because it is related from base\_x & base\_y

**C-s:**

1. Class [Village](#_Village).
2. Class [AttackQueue](#_AttackQueue).
3. Class [RequestManager](#_RequestManager).

### Village

Representation of a single village.

R-s:

1. Holds info about particular village.
2. Lives in **Map**.
3. Updates itself from [**AttackReport**](#_AttackReport).
4. Used by **Attack** to construct an attack.
5. Fields:

* Coordinates (tuple, (x,y))
* Id (int)
* Resources remained (default = None)
* Last visited (time, default=None)
* Mine levels (tuple, (wood, clay, iron), default = None)
* Resources h/rates (adjusted by Bonus, tuple, default=None)
* Resource looted (total/per visit). ({“total”: int, “per\_visit”: [(time, int), …]})

1. Methods:

* Constructor:

\_\_init\_\_(self, coordinates=(x,y), id=int, bonus=None)

* Get\_rates\_mapping(self) : returns list [(level, income), …], see game Help page
* Set\_h\_rates(self, tuple): accepts tuple of 3 values (wood level, clay level, iron level), sets self.h\_rates
* Estimate\_haul(self, time\_of\_next\_visit): returns estimated haul capacity on the next visit basing on self.h\_rates & self.t\_of\_last\_visit.
* Update\_stats(self, AttackReport obj): updates self.mine\_levels, self.h\_rates, self.t\_of\_last\_visit, self.resource\_remained basing on the data from AttackReport object.

C-s:

1. Class [Map](#_Map)
2. Class [AttackReport](#_AttackReport)
3. Class [Attack](#_Attack).

### AttackReport

Representation of a single attack.

R-s:

1. Holds info about particular attack.
2. Inited by **ReportBuilder** from individual HTML file.
3. Passed by **AttackQueue** to particular **Village** to update **Village** info.
4. Fields:

* Status (red, yelllow, green)
* Time (t\_of\_attack)
* .mine\_levels
* .remaining\_capacity
* .looted\_capacity

1. Methods:

* Constructor (self, t\_of\_attack, str\_html)
* Getters for each field (‘re’ parsing from given html string)

C-s:

1. Class [AttackQueue](#_AttackQueue).
2. Class [Village](#_Village).
3. Class [ReportBuilder](#_ReportBuilder).

### AttackQueue

Representation of priority-based queue of Village objects.

R-s:

1. Stores a sequence of **Villages**, sorted by a distance to base point (nearest=first).
2. Asks **Map** for a list of **Villages** with particular distance limit.
3. Responds to **Bot** with a **get\_next\_village()** method (1st village in queue to attack).
4. Updates **Villages** with a list of given **AttackReports**
5. Updates **Map** with a list of self.villages on application close()
6. Methods:

* **\_\_init\_\_(**self, x, y, request\_manager, map\_depth, queue\_depth, farm\_frequency, capacity\_threshold, mapfile, farm\_radius):

Self.map = Map(x, y, request\_manager, depth=map\_depth, mapfile=mapfile)

Self.radius = farm\_radius

Self. depth = queue\_depth

Self.village\_rest = farm\_frequency \* 3600

Self.threshold = capacity\_threshold

Self.villages = self.map.get\_villages\_in\_distance(self.radius)

Self.queue = []

Self.build\_queue()

* Def **valid\_for\_farm**(village):

If village.remaining\_capacity > self.threshold

Elif not remaining\_capacity and not village.last\_visited

Elif not remaining and (time\_gmt – villa.last\_visited) > self.rest:

Return True

Else: false

* Def **build\_queue**():

Self.queue = sorted\_by\_distance(villa in self.villages if valid\_for\_farm(villa))

* Def **is\_attack\_possible**(village, troops\_map):

For troop in troops\_map:

Gets village distance

Gets troop speed

Estimates arrival from gmt\_t\_now

Asks village to estimate capacity

Troops needed = round(estimated\_capacity/troop\_haul)

Checks if troops\_map[troop][count] >= Troops needed

Returns {troop, count}

Else: returns false

* Def **get\_next\_attack\_target**(self, troops\_map):

If self.queue:

High\_priority = self.queue[:1]

Self.is\_attack\_possible(high\_priority)?

Del high\_priority from self.queue

Return (coords, {troop:count})

Else:

Next\_priorities = self.queue[1:self.depth]

If next\_priorities:

For each next\_priority ask is\_attack\_possible, etc.

Return None

* Def **update\_villages**(self, reports):

For report in reports:

Self.villages[report[coords]].update\_stats(report)

Self.build\_queue()

C-s:

1. Class [Village](#_Village).
2. Class [Map](#_Map).
3. Class [Bot](#_Bot).
4. Class [AttackReport](#_AttackReport).

### Attack

Component responsible for sending a single attack.

R-s:

1. Makes a sequence of calls to **RequestManager**  to send an attack:

* Get rally point view (needs to extract few hidden fields from response-HTML and pass them as additional cookies to RequestManager).
* Get confirmation screen (needs to extract token values from hidden fields)
* Send attack request.

1. Accepts **village** obj. & **Bot** instance.
2. Respond with a **.is\_valid()** method which makes a decision, whether the given village could be attacked now (basing on Bot.troops mapping and village estimated capacity)
3. Calculates arrival and back-time.
4. Returns arrival & back time when sends an attack.

C-s:

1. Class [RequestManager](#_RequestManager).

2. Class [Bot](#_Bot).

3. class [Village](#_Village).

### ReportBuilder

Component for building AttackReports.

R-s:

1. Calls **RequestManager** for a report summary page (HTML)
2. Extracts all URLs for “(new)” reports.
3. Calls **RequestManager** for a particular report (URL or id)
4. Inits **AttackReport**
5. Responds to **Bot** with a list of **AttackReports.**

C-s:

1. class [RequestManager](#_RequestManager).
2. class [AttackReport](#_AttackReport).
3. class [Bot](#_Bot).

### AttackObserver

Component responsible for observing troops arrival.

R-s:

1. Observes registered time-values (arrival time & backtime) on a constant basis (should be subclassed from Thread to not block parent process).
2. When current\_time > than any of registered values, notifies **Bot** by setting one of its flags to True: .new\_reports (arrival time) & .needs\_refresh (backtime).
3. New time values are added by **Attack** instances.
4. **Bot’s** component.

C-s:

1. class [Attack](#_Attack).

2. class [Bot](#_Bot).

### TroopsManager

Component responsible for calculating troops count. (may be extended to calculate also resources, etc.)

R-s:

1. Make call to **RequestManager** for a village overview.
2. Parses response-HTML to get mapping of troops that are currently in a village.
3. Respond to a **Bot** with mapping of troops.

C-s:

1. Class [Bot](#_Bot).
2. Class [RequestManager](#_RequestManager).

### RequestManager

Component responsible for sending requests, decompressing raw data & CAPTCHA verification.

R-s:

1. Forms and sends requests to a server
2. Decompresses and decodes raw response.
3. Verifies CAPTCHA presence in a HTML response files.
4. Responsible for **Notification** process if CAPTCHA is found.
5. Responsible for getting base cookies (village\_id, mobile, cid, sid) from browser.
6. Invoked by **ReportBuilder**.
7. Invoked by **Attack**.
8. Invoked by **Map**.

C-s:

1. Class [ReportBuilder](#_ReportBuilder).
2. Class [Attack](#_Attack).
3. Class [Map](#_Map).