Snakes On a Boat
Design Document

Martin Bertrand
Jordan Brobyn
Michael Delong
Stephanie Hurtado
Nicholas Presta
Joey Vansteenkiste

Table of Contents

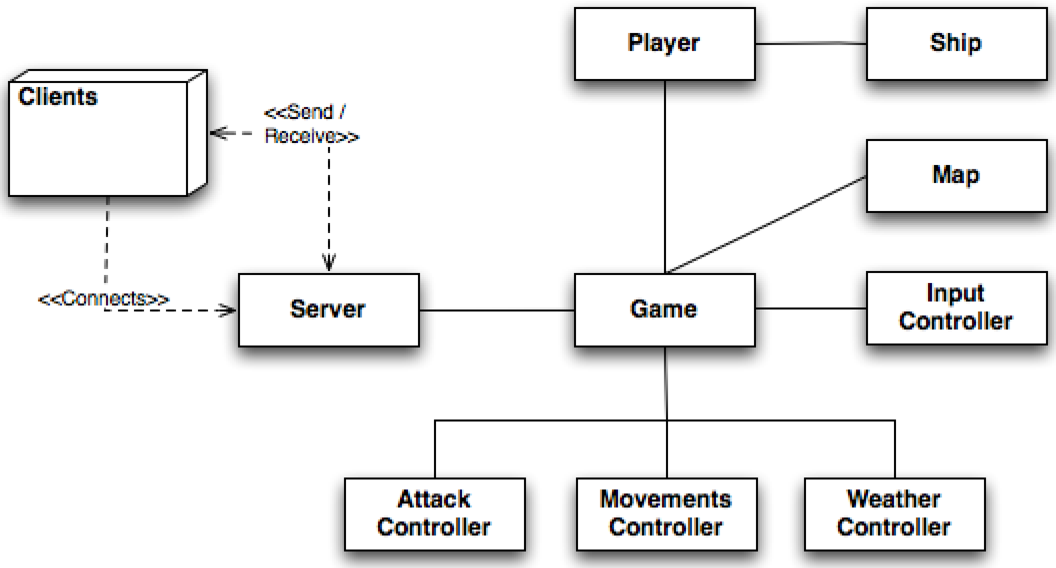
Introduction

The purpose of this document is to clearly communicate the design decisions taken, and why these decisions will be effective for the purpose of developing a game server, which will henceforth be known as "Snakes on a Boat". The design document will cover decisions such as what programming language(s) have been chosen and why, class design, functional design, and architectural strategies. This document is intended for developers of the game server. The document will use technical terms involving functions used, and classes to be implemented during the programming phase. The game server will be written in Python, utilizing *SQLite* to store and retrieve data, and will be able to send/receive data from multiple clients at a time through a multi-threading architecture.

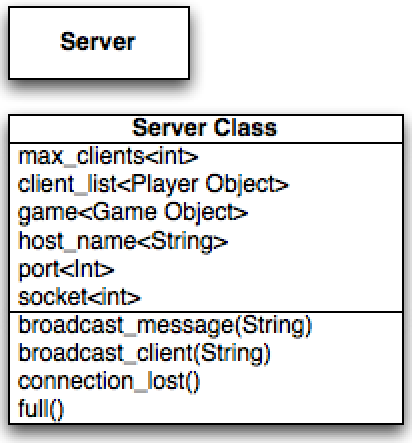
Design implementation will be based on previous requirement documents and use cases created by all groups. However, only a small subset of features will be developed due to time constraints and intentional simplicity of the game.

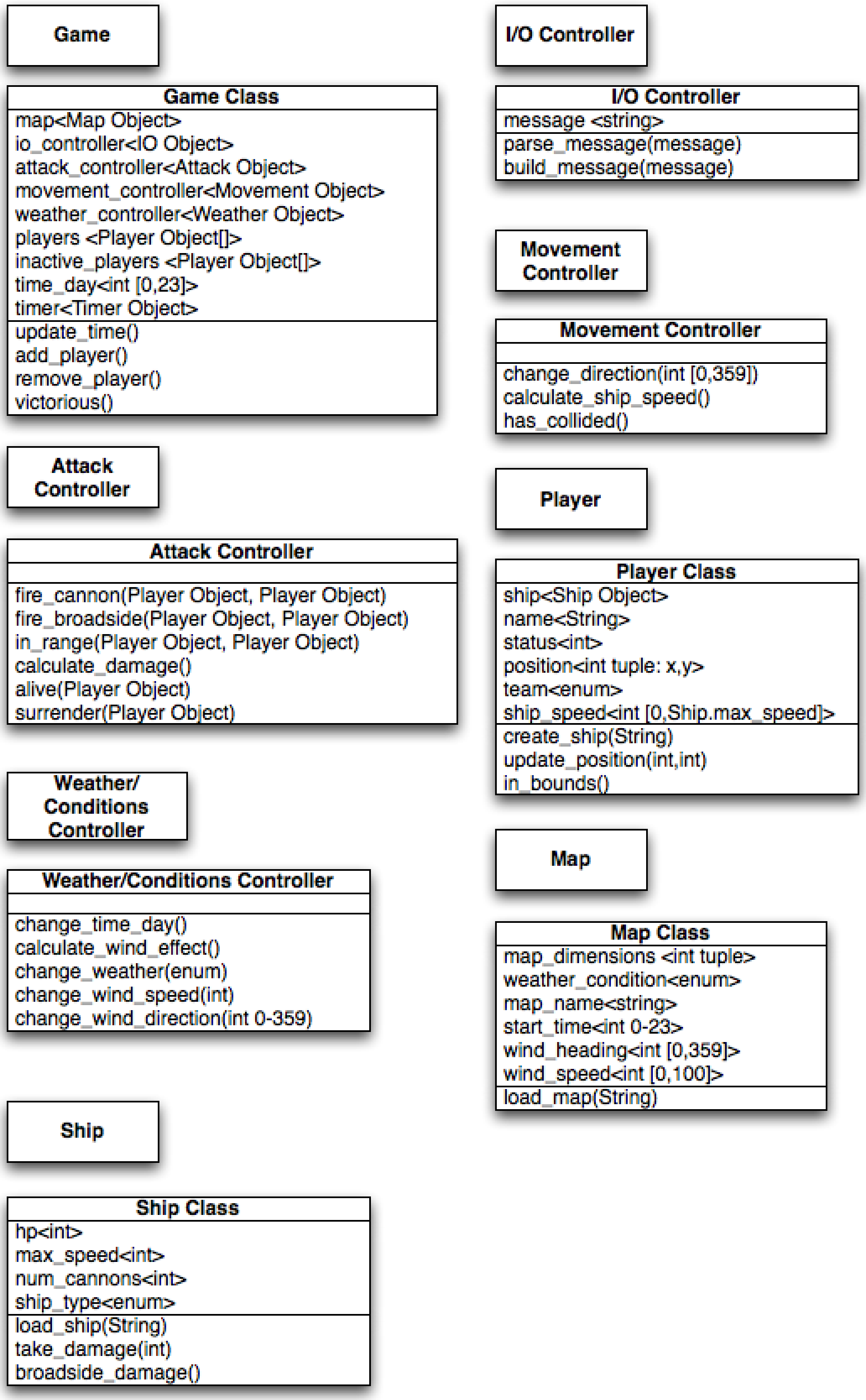
Design Overview

The server will receive and maintain multiple clients by creating a thread for each incoming client. Each thread will listen continuously for messages sent from its respective client.
Messages received will then be sent to the Game Class to be parsed and processed by the controller functions contained within the Game Class, depending on the received message. The Ship Class will maintain client ship information, which will be updated based on events created by the clients during game play.
A Player Class will maintain the Ship Class objects as well as hold the client's name and current status (ex. destroyed, surrendered). After data has been processed by the server, the server will return updated information to all clients connected. The clients should then update any relevant data with the new information received. The server must be able to check for mistakes, or cheating attempts, that may give unfair advantage to certain clients. Here is a general overview of the server classes and how they are related.



Detailed Design




Game
Game Class
map<Map Object>
io_controller<IO Object>
attack_controller<Attack Object>
movement_controller<Movement Object>
weather_controller<Weather Object>
players <Player Object[]>
inactive_players <Player Object[]>
time_day<int [0,23]>
timer<Timer Object>
update_time()
add_player()
remove_player()
victorious()
Attack Controller
Attack Controller
fire_cannon(Player Object, Player Object)
fire_broadside(Player Object, Player Object)
in_range(Player Object, Player Object)
calculate_damage()
alive(Player Object)
surrender(Player Object)
Weather/ Conditions Controller
Weather/Conditions Controller
change_time_day()
calculate_wind_effect()
change_weather(enum)
change_wind_speed(int)
change_wind_direction(int 0-359)
Ship
Ship Class
hp<int>
max_speed<int>
num_cannons<int>
ship_type<enum>
load_ship(String)
take_damage(int)
broadside_damage()
I/O Controller
I/O Controller
message <string>
parse_message(message)
build_message(message)
Movement Controller
Movement Controller
change_direction(int [0,359])
calculate_ship_speed()
has_collided()
Player
Player Class
ship<Ship Object>
name<String>
status<int>
position<int tuple: x,y>
team<enum>
ship_speed<int [0,Ship.max_speed]>
create_ship(String)
update_position(int,int)
in_bounds()
Map
Map Class
map_dimensions <int tuple>
weather_condition<enum>
map_name<string>
start_time<int 0-23>
wind_heading<int [0,359]>
wind_speed<int [0,100]>
load_map(String)

Class Descriptions

Server Class

This class will accept connections from clients, and send/receive messages to and from multiple clients. Messages received from clients are passed to the Game instance. Messages that are to be sent to clients will be passed to this class from the Game instance. The map name and maximum number of players will be passed to the server through the command line.
The Server class is the main entry point for the "Snakes On a Boat" server application.

Variables

- game <Game instance>: represents an instance of a Game object
- client_list <list>: list of socket connections maintained by server
- max_clients<int>: maximum number of players allowed by the server
- hostname <String>: hostname of server
- port <int>: port server will listen on for connections
- socket<int>: socket the server uses for communication

Methods

- broadcast_message(string message): used to send updated messages to all clients
- broadcast_client(string message): used to send individual messages to a specific client
- connection_lost(): remove a disconnected client from the client_list
- full(): checks if the maximum number of players allowed in the game has been reached; returns a boolean - true or false.

Game Class

The Game class is responsible for overseeing the overall structure and flow of the game. It will be created by the Server class when the program is launched.

Variables:

- io_controller <IO Controller>: instance of input/output controller class; handles the parsing of incoming messages and the constructing of outgoing messages.
- attack_controller <Attack Controller>: instance of attack controller class; handles calculations and results of player attacks on other players.
- movement <Movement Controller>: instance of movement controller class; handles all calculations and updates related to ship movement.
- weather <Weather Controller>: instance of weather controller class; handles all changes and updates in weather conditions during the game.
- active_players <list[Player]>: list of Player objects, representing connected players who are currently active in the game. Each element in this list will represent a team, and inside this sub-list, the players who are part of this team will be stored.
- inactive_players <list[Player]>: list of Player objects, representing connected players who are currently inactive in the game (have been defeated or surrendered).
- map <Map>: instance of Map class; this will be the playing area on which the game

takes place.
- time_day <int [0, 23]>: represents time of day; [0, 11] represents daytime, and [12, 23] represents nighttime.
- timer <Timer>: uses Timer object (implementation of a thread) from the Python standard library to keep track of the running time of the game.

Methods:
- update_time(): executed within the Timer thread; calls changeTimeDay() from the "weather" object every 12 minutes.
- add_player(int id): adds a newly connected player to the active_players list, and assigns the player a random location on the map.
- remove_player(Player Object): removes player from active_players list and adds to the inactive_players list. This method will be called when a player is defeated.
- victorious(): checks if one team is victorious; this occurs if one of the team lists is empty. Return true or false.
- get_player(int id): Returns a player object from the active_players list, given an integer (from the client).

IO Controller

The input/output controller will parse messages received by the server, and call the appropriate use case controller. This class will also build update messages that will be sent back to client(s).
- Methods:
  - parse_message(string message) - parse message received from client (Server class passes it to Game class, Game class passes to IO Controller). Returns a dictionary.
  - build_message(dict message) - create message to send to client; pass to Server class when done. Returns a string.

Attack Controller

Encapsulates everything needed for players to perform attack functions. This includes firing cannon(s), dealing damage, and updating ship status.
- Methods:
  - fire_cannon(Player Object, Player Object) - fires a single cannon on the enemy ship; integer represents the player ID to fire on.
  - fire_broadside(Player Object, Player Object) - fires all cannons on the same side as the enemy ship; integer represents the player ID to fire on.
  - in_range(Player Object, Player Object): checks if the targeted enemy ship is within firing range; returns true or false; the integer parameter represents the enemy ship number. This will be called by the fire_cannon() and fire_broadside() methods.
  - calculate_damage(): calculates damage done to the enemy ship; this will be called by the fire_cannon() and fire_broadside() methods.
  - alive(Player Object): checks if a ship's hit points have reached 0; return true or

false.
    - surrender(Player Object): player with the specified integer ID surrenders, resulting in hit points reaching 0 and player being removed from the game.
    - update_max_speed(Player Object): updates the maximum speed the player can reach based on damage done

Movement Controller

Encapsulates everything needed to move and change speed. This includes updating speed, changing a heading, and turning. This class will utilize the universal turning model devised by Daniel R. This controller will access speed and position variables that will be stored in the Player class.
Methods:

- change_direction(int [0, 359]) - called when client requests to turn ship; the turn angle is represented as an angle in clockwise direction from due north in degrees: (0=N, 90=E, 180=S, 270=W). This will utilize Dan R's turning model to calculate and return the endpoint of the turn. This method will update the position of the ship (which is a variable inside the Player class), to the endpoint coordinates resulting from the turn.
- calculate_ship_speed() - this method would be called within the change_direction() method to calculate the ship's change in speed after a turn. The Weather Controller would also call this method to update speed after a change in wind speed/direction. This method will also be called if the user initiates a speed change.
- Optional: has_collided() - checks for a collision with land masses or other ships - returns true or false

Weather/Conditions Controller

Encapsulates impact on ship and player status due to weather conditions such as rain, snow and fog.
Methods:

- change_time_day() - toggles the day/night status of the map each time it is called. This will be called by the Game class every 12 real-time minutes.
- calculate_wind_effect(): calculates and returns the wind factor on ship speed as a float
- calculate_weather_effect(): calculates and returns the weather factor on ship speed as a float
- Optional: change_wind_speed(int [0,100] or float[0, 1]) - calls calculate_ship_speed() after wind speed is updated
- Optional: change_wind_direction(int [0, 359]) - changes wind direction; the new direction is represented as an angle in clockwise direction from due north in degrees: (0=N, 90=E, 180=S, 270=W); calls calculate_ship_speed() after wind direction is updated.
- Optional: set_light(boolean) - turns a ship's light on or off, improving visibility

Map Class

This class holds all the information for the Map object stored in the Game class. The data for this class will be loaded from the SQLite map database.

Variables:

- weather_condition <enum>: Weather condition, persistent throughout the game. This will be one of CLEAR, RAIN, SNOW, or FOG.
- map_name <String>: name of the map
- start_time <int>: An integer between 0 and 23 (0 is midnight and 23 is 11pm).
- map_dimensions <tuple(int, int)>: contains two integer values: the first represents the vertical size of the map (x value), the second represents the horizontal size of the map (y value).
- Optional: land_masses <list[tuple(Vertices)]>: each element will be a tuple holding Vertices which is a list consist of at least two x and y coordinates representing a land mass.
- wind_heading <int>[0, 359]: represents the heading of the wind; the heading is represented as an angle in clockwise direction from due north in degrees: (0=N, 90=E, 180=S, 270=W)
- wind_speed <int> [0,100]: speed of the wind; 0 is no wind, 100 is maximum wind speed.

Methods:

- load_map(String): loads the map data from the map database, and stores the results in the class variables. If the map is not found in the database, an error will be generated and the program must exit.

Player Class

This class is used to store data for each player connected to the game. The class will keep track of ship position and status.

Variables:

- ship <Ship>: instance of ship object that belongs to the player
- name <String>: screen name of connected player
- team <enum>: team player belongs to
- player_id <int>: ID number of player
- status <enum>: DEFEATED, WON, SURRENDERED, PLAYING
- position <tuple(int, int)>: position of player ship represented by coordinates (x, y)
- ship_speed <int> [0, Ship.max_speed]: represents speed of the ship

Methods:

- create_ship(String): creates the ship selected by the player
- update_position(int, int): updates the position of the ship on the map
- in_bounds(): checks if the ship has gone past the boundaries of the map

Ship Class

This class holds the properties of the ship a player will be using. When a ship is created, the properties will be read from the SHIPS SQLite table and stored in the class variables.

Variables

- hit_points <int> [0-100]: represents “health” of the ship, 0 means the ship has been destroyed, 100 is perfect condition. Default value is 100.
- max_speed <int> [0-100]: represents the maximum speed of the ship. This can change if damage is done to the ship. The maximum speed attainable will also depend on the ship type.
- num_cannons <int>: number of cannons aboard the ship. The more cannons, the more damage the ship can deal out. The number of cannons is dependent on the ship type.
- ship_type <enum>: represents the type of ship
- name <string>: A string name representing the ship.

Methods

- load_ship(String): loads and stores the data for the specified ship from the database
- take_damage(int): reduces ship hit points by the specified number of damage points taken
- broadside_damage(): returns an integer specifying the damage that will be caused by a broadside from this ship. This will depend on the number of cannons the ship has

Testing

In Python, it is very easy to write unit tests using the built-in unit test framework (http://docs.python.org/library/unittest.html). This allows each module to be tested individually by creating a test suite that runs the various test cases. The test cases should be based on use cases, such as moving a ship, firing a ship, and making sure the appropriate actions occur, such as changing position, and losing HP. Each class should have a unit test to test basic functionality and correctness.

Due to the nature of this application, a way is needed to test basic networking capabilities and the ability to serve multiple clients simultaneously. An extremely simple client will need to be written that can be used to simulate a client connecting to the server. The unit test framework can be used to test the responses from the server.

It may also be possible to utilize pylint (http://pypi.python.org/pypi/pylint) to check the code for common errors and coding standards.

Summary

Snakes on a Boat is a client-server game simulating a series of naval combats. The server will mostly be responsible for relaying messages to and from clients. Apart from relaying messages, the server will create new games, accept connections and will overlook the general flow of the game, sending appropriate messages to the clients. The application will be written in Python, a powerful and easy to use object oriented language.

Appendix A: Design Decisions

- Using Python 2.6.6 to implement the application
    - Development time is reduced compared to using a language such as Java or C
        - Less is more. Java is too verbose.
    - Code is clear, concise, and readable (reads like pseudo-code)
    - Rich data types (dictionaries, lists, tuples, sets, stacks, queues)
        - Prevents "home grown" data structures that are often implemented poorly
    - Methods/Functions can return multiple values
        - Useful in some circumstances and aids in explicitness and readability
    - Keyword arguments (adds to explicitness and readability)
    - Batteries are included (writing base libraries is rare)
        - Networking, database, parsing, etc, are included and well documented
    - Exceptions are easily extensible (no checked exceptions!)
        - Easy to catch specific exceptions, creating custom exceptions is easy
- Utilizing SQLite 3.6.12 as a database back end to store ship and map data
    - Simple, flat-file database
        - Does not require a complex database system
    - Using Elixir (http://elixir.ematia.de/trac/wiki) as a declarative layer for SQLAlchemy's Object-Relational Mapping.
        - This allows the separation of business logic from data access.
- Storing map, weather, and other game data on the server.
    - This prevents the programming of clients for the purpose of obtaining an unfair advantage over other players.
    - Map Data: Name, size (height/width), weather effect, max number of players

Appendix B: Style Guide

Heavily Based on Python's PEP 8 document: http://www.python.org/dev/peps/pep-0008/. In essence, The PEP 8 guidelines have been taken and simplified (removing uncommon cases) and changed to suit this project needs. For a more complete explanation of each section, read the PEP 8 guideline.

Code Layout

- Indentation: Use 4 spaces (not tabs!)
- Tabs should not be used at all.
- Maximum Line Length: All lines should be limited to 90 characters. In Python, there is an implied line continuation between brackets, }{, parentheses,)(, and braces,][, meaning there's no need to worry about breaking lines in these constructs (lists, tuples, dictionaries, etc). In the event one needs to break long lines, use the backslash operator, \, and make sure to indent appropriately.
- Breaking Binary Operators: When breaking a line after a binary operator (or, and, +, -, *, /, etc), it is preferable to break after the operator, not before.
- Blank Lines: Separate classes and top-level functions with two blank lines. Methods within a class should be separated by one blank line.

Imports

- Imports should be on separate lines unless importing many modules from one name space.
- Imports must appear at the top of a file, after file/module comments.
- Order of imports:
    - Standard libraries: Libraries that ship with Python (sys, os, re, etc) should be first
    - Officially supported or recognized libraries should be next. These include things like *Django*, *Beautiful Soup*, and other big name or well known libraries
    - Custom libraries: Modules written for inclusion by the design team should be last.

White Space in Expression and Statements

- Avoid extra white space:
    - Immediately inside parentheses, brackets or braces.
    - Immediately before a comma, semi-colon, or colon.
    - Immediately before the parenthesis in a function call.
    - Immediately before the parenthesis in index/list slicing.
    - Around assignment operators (one space on either side is plenty).
    - Binary operators should have single spaces on either side of them.
    - The equals sign should not have any spaces around it when specifying default parameters or named arguments in a function call.

Comments

- Comments should be complete sentences.
- Block Comments:
    - Apply to all code that follows in the same indentation level
    - Should start with a hash (#) followed by one space.
- Inline comments:
    - Use inline comments (comments that start on the same line as code) sparingly.
    - Inline comments should start with a hash (#) and a single space.

Document Strings

- Document strings (commonly referred to as "docstrings") are essential when programming in Python.
- All public modules, functions, classes, and methods must have docstrings.
- Private methods do not need a docstring, but must still include a comment explaining the purpose of the method.
- Docstrings must appear after the `def` line in classes, modules, methods, functions.
- Docstrings start and end with three double quotes - " " ".

Naming Conventions

- Python modules (files that contain code that can be imported) should be all lowercase but may contain underscores if it helps with readability.
- In all cases, classes should use CapsWords convention.
- Internal, or private classes should start with an underscore (followed by CapsWords name).
- Exceptions are classes and as such, follow the CapsWords convention.
- Function names should be all lowercase, with underscores used to separate words.
- For instance methods, the first argument should always be named `self`.
- For class methods, the first argument should always be named `cls`.
- Method names and instance variables should always be all lowercase with underscores used to separate words.
- Use one leading underscore to signify private instance methods or variables.
- Constants are declared in block capitals..

General Guidelines

- All "new style" classes should inherit from `object` if they do not inherit from something else.
- Only specific exceptions may be caught. Python allows exceptions to be grouped in a try/except block. A bare except should not be used without an exception type.
- Limit the try/except block to the minimum amount of code necessary . This helps with debugging.
- Comparing things to None should be done using `is` or `is not`, never == or `!=`.
- Don't compare Boolean values to True or False.

- Use the fact that containers (lists, tuples, dictionaries) are False when empty and True when they have items.. This allows: `if not foo` instead of `if len(foo) == 0`.