Oval marking algorithm that integrates other algorithms and integrates mathematical and/or logical concepts shown on pages (3-5)
Rectangle marking a segment of code I developed that represents an abstraction is shown below on page (1)

```cpp
#include "CommandProcessor.h"
#include "PrintValueJob.h"
#include "ReminderSystem.h"
#include "TextReminder.h"
#include "SoundReminder.h"
#include "TextReminderJob.h"
#include "SoundReminderJob.h"
#include "ReminderListJob.h"
int main()
{

      /*
      First we need to instantiate the command processor so it can start
processing commands of the format described in the file Command.h
      */
      CommandProcessor cmdProcessor;


      std::unique_ptr<CommandJob> soundReminderJob =
std::make_unique<SoundReminderJob>();
      cmdProcessor.AddJobForCommand(soundReminderJob);

      std::unique_ptr<CommandJob> reminderListJob =
std::make_unique<ReminderListJob>();
      cmdProcessor.AddJobForCommand(reminderListJob);

      std::unique_ptr<CommandJob> textReminderJob =
std::make_unique<TextReminderJob>();
      cmdProcessor.AddJobForCommand(textReminderJob);

      pReminderSystem = new ReminderSystem();
      pReminderSystem->Start();

      std::cout << "You can set your reminders by typing below..." <<
std::endl;

      while (true) {
            std::string in;
            std::cin.ignore();
            std::getline(std::cin, in);
            in = "-" + in;
            if (!cmdProcessor.ProcessCommand(in))
                  std::cout << "Failed to process command : " << in <<
std::endl;
      }



      getchar();
    return 0;
}

#include "Command.h"

Command::Command()
{
      commandTitle.clear();
      commandArgs.clear();
      commandId = -1;
}

#include "CommandJob.h"
#include <algorithm>
CommandJob::CommandJob(const std::string name)
{
      commandName = name;
      std::transform(commandName.begin(), commandName.end(),
commandName.begin(), ::tolower); // convert command name to lowercase
}

const std::string& CommandJob::GetCommandName()
{
      return commandName;
}

#include <algorithm>
#include "CommandProcessor.h"

int CommandProcessor::commandCount = 0;

CommandProcessor::CommandProcessor()
{
      rawCommands.clear();
      parsedCommands.clear();
      commandJobs.clear();
      commandCount = 0;
}

void CommandProcessor::AddJobForCommand(std::unique_ptr<CommandJob>& job)
{
      commandJobs.insert(std::make_pair(job->GetCommandName(),
std::move(job)));
}


bool CommandProcessor::ProcessCommand(std::string rawCmd)
{

      auto findInString = [&](const std::string& haystack, char toFind,
const size_t startIndex = 0) {
            if (startIndex < 0 || startIndex > haystack.size() + 1)
                  return std::string::npos;

            for (size_t i = startIndex; i < haystack.size(); i++) {
                  if (haystack[i] == toFind)
                        return (size_t)i;
            }

            return std::string::npos;
      };

      auto getSubstringString = [&](const std::string& haystack, size_t
start, size_t end) {
            if (start < 0)
                  throw std::invalid_argument("start");

            if (end > haystack.size() + 1)
                  throw std::invalid_argument("end");

            std::string resultString = "";

            for (size_t i = start; i < end; i++) {
                  resultString += haystack[i];
            }
            return resultString;
      };


      if (rawCmd.empty()) // Command string is empty
            return false;

      auto dashChar = findInString(rawCmd,'-', 0);


      if (dashChar == std::string::npos)
            return false; // Could not find the beginning '-'

      auto nameEndingChar = findInString(rawCmd, ' ', dashChar);

      if (nameEndingChar == std::string::npos)
            nameEndingChar = rawCmd.length();

      rawCommands.push_back(rawCmd);

      Command newCmd;
      newCmd.commandTitle = getSubstringString(rawCmd, dashChar + 1,
nameEndingChar);
      std::transform(newCmd.commandTitle.begin(),
newCmd.commandTitle.end(), newCmd.commandTitle.begin(), ::tolower); //
Convert the name to lowercase as the spec defines.
      newCmd.commandId = commandCount; // Give command a unique identifier
      commandCount++; // Increase command count.

      auto argSearchIndex = nameEndingChar + 1; // Search for args where we
found the name.

      while ((argSearchIndex = findInString(rawCmd, '$', argSearchIndex))
!= std::string::npos) {

            // Found another argument

            auto argEnd = findInString(rawCmd, ' ', argSearchIndex);

            if (argEnd == std::string::npos) {
                  argEnd = rawCmd.size(); // If we can't find the
separating space, this argument is the last characters in the string.
            }

            auto argStart = argSearchIndex + 1; // Go past the $ prefix.

            auto argument = getSubstringString(rawCmd, argStart, argEnd);
// Grab only the argument out of the text, without the $

            newCmd.commandArgs.push_back(argument); // Add the argument to
the array

            argSearchIndex = argStart; // Next iteration start searching
after this argument.
      }

      return ProcessCommand(newCmd); // Pass parsed command to the actual
method that executes the commands function.
}

bool CommandProcessor::ProcessCommand(const Command & cmd)
{
      if (commandJobs.count(cmd.commandTitle) == 0)
            return false; // Could not find a job that matches the target
command title.

      auto& cmdJob = commandJobs[cmd.commandTitle];

      auto res =  cmdJob->OnCommand(cmd);

      return res;
}

#include "PrintValueJob.h"
#include <iostream>
#include <string>
PrintValueJob::PrintValueJob() : CommandJob::CommandJob("PrintArgs")
{

}

bool PrintValueJob::OnCommand(const Command & cmd)
{
```

```cpp
	if (cmd.commandArgs.empty())
		return false;

	for(auto& arg : cmd.commandArgs) {
		std::cout << arg << std::endl;
	}

	return true;
}
#include "Reminder.h"
#include <ctime>
#include <iomanip>
Reminder::Reminder()
{
	running = false;
}

void Reminder::Start(std::chrono::nanoseconds timeTillDone)
{
	running = true;
	startTime = std::chrono::steady_clock::now();
	endTime = startTime + timeTillDone;
}

void Reminder::Stop(bool didWholeTimeElapse)
{
	running = false;

	if (!didWholeTimeElapse)
		return;
	else {
		this->OnDone(endTime - startTime);
	}
}

bool Reminder::IsRunning()
{
	return running;
}

std::chrono::time_point<std::chrono::steady_clock> Reminder::GetStartTime()
{
	return startTime;
}

std::chrono::time_point<std::chrono::steady_clock> Reminder::GetEndTime()
{
	return endTime;
}



time_t steady_clock_to_time_t(std::chrono::steady_clock::time_point t)
{
	return
std::chrono::system_clock::to_time_t(std::chrono::system_clock::now() +
std::chrono::duration_cast<std::chrono::system_clock::duration>(t -
std::chrono::steady_clock::now()));
}

std::string
Reminder::ConvertTimeToString(std::chrono::time_point<std::chrono::steady_c
lock> t)
{
	std::time_t time = steady_clock_to_time_t(t);
	std::tm timetm = *std::localtime(&time);
	return std::string(std::ctime(&time));
}


std::chrono::duration<double> Reminder::GetTimeLeft()
{
	return endTime - std::chrono::steady_clock::now();
}
#include "ReminderListJob.h"
#include "ReminderSystem.h"
#include "SoundReminder.h"
ReminderListJob::ReminderListJob() :
CommandJob::CommandJob("ListReminders")
{

}

bool ReminderListJob::OnCommand(const Command & cmd)
```

```
{
	if (cmd.commandArgs.size() > 0)
		return false;

	auto& reminderList = pReminderSystem->GetReminders();

	for (auto& reminder : reminderList) {
		std::cout << reminder->GetDescription() << std::endl;
	}

	return true;
}
#include "ReminderSystem.h"
#include <thread>
#include <chrono>
ReminderSystem::ReminderSystem()
{
	reminders.clear();
}

void ReminderSystem::AddReminder(std::unique_ptr<Reminder>& reminder,
std::chrono::nanoseconds timeTillDone)
{
	reminder->Start(timeTillDone);
	reminders.push_back(std::move(reminder));
}

void ReminderSystem::Start()
{
	std::thread tickThread(&ReminderSystem::Tick, this);
	tickThread.detach();
}

std::vector<std::unique_ptr<Reminder>>& ReminderSystem::GetReminders()
{
	return reminders;
}

void ReminderSystem::Tick()
{
	while (true) {
```

```
            for (auto& reminder : reminders) {
                  if (reminder->IsRunning()) {
                        if (reminder->GetTimeLeft().count() <= 0) {
                              reminder->Stop(true); // Alert reminder that
it should trigger. Also mark the reminder as not running.
                        }
                  }
            }

            std::this_thread::sleep_for(std::chrono::seconds(1)); //
Maximum timer accuracy of 1 second.
      }
}


ReminderSystem* pReminderSystem;
#include "SoundReminder.h"
#include <windows.h>
#include <sstream>
SoundReminder::SoundReminder()
{
}

void SoundReminder::OnDone(std::chrono::duration<double> timeElapsed)
{
      Beep(523, 500);
}

std::string SoundReminder::GetDescription()
{
      std::stringstream stream;

      stream << "SoundReminder was started on " <<
ConvertTimeToString(GetStartTime()) << " and will end on " <<
ConvertTimeToString(GetEndTime()) << ".";

      return stream.str();
}

#include "SoundReminderJob.h"
#include "ReminderSystem.h"
#include "SoundReminder.h"
```

```
SoundReminderJob::SoundReminderJob() :
CommandJob::CommandJob("AddSoundReminder")
{

}

bool SoundReminderJob::OnCommand(const Command & cmd)
{
	if (cmd.commandArgs.size() < 2)
		return false;

	auto timeScale = cmd.commandArgs[0];
	auto timeAmmount = atoi(cmd.commandArgs[1].c_str());

	std::unique_ptr<Reminder> soundRemind =
std::make_unique<SoundReminder>();

	if (timeScale == "seconds") {
		pReminderSystem->AddReminder(soundRemind,
std::chrono::seconds(timeAmmount));
	}
	else if (timeScale == "minutes") {
		pReminderSystem->AddReminder(soundRemind,
std::chrono::minutes(timeAmmount));

	}
	else if (timeScale == "hours") {
		pReminderSystem->AddReminder(soundRemind,
std::chrono::hours(timeAmmount));

	}
	else if (timeScale == "days") {
		pReminderSystem->AddReminder(soundRemind,
std::chrono::hours(24) * timeAmmount);

	}
	else {
		return false;
	}

	std::cout << "Added a new SoundReminder that will go off in " <<
timeAmmount << " " <<  timeScale << "!" << std::endl;
```

```
	return true;
}
#include "TextReminder.h"
#include <sstream>
#include <windows.h>
TextReminder::TextReminder(std::string text) : Reminder()
{
	textToShow = text;
}

void TextReminder::OnDone(std::chrono::duration<double> timeElapsed)
{
	///std::cout << textToShow << std::endl;
	MessageBoxA(NULL, textToShow.c_str(), "Text Reminder", NULL);
}

std::string TextReminder::GetDescription()
{
	std::stringstream stream;

	stream << "TextReminder with message " << "'" + textToShow << "' was
started on " << ConvertTimeToString(GetStartTime()) << " and will end on "
<< ConvertTimeToString(GetEndTime()) << ".";

	return stream.str();
}
#include "TextReminderJob.h"
#include "ReminderSystem.h"
#include "SoundReminder.h"
#include "TextReminder.h"
TextReminderJob::TextReminderJob() :
CommandJob::CommandJob("AddTextReminder")
{

}

bool TextReminderJob::OnCommand(const Command & cmd)
{
	if (cmd.commandArgs.size() < 3)
		return false;
	auto timeScale = cmd.commandArgs[0];
	auto timeAmmount = atoi(cmd.commandArgs[1].c_str());
	auto txt = cmd.commandArgs[2];

	std::unique_ptr<Reminder> soundRemind =
std::make_unique<TextReminder>(txt);

	if (timeScale == "seconds") {
		pReminderSystem->AddReminder(soundRemind,
std::chrono::seconds(timeAmmount));
	}
	else if (timeScale == "minutes") {
		pReminderSystem->AddReminder(soundRemind,
std::chrono::minutes(timeAmmount));

	}
	else if (timeScale == "hours") {
		pReminderSystem->AddReminder(soundRemind,
std::chrono::hours(timeAmmount));

	}
	else if (timeScale == "days") {
		pReminderSystem->AddReminder(soundRemind,
std::chrono::hours(24) * timeAmmount);

	}
	else {
		return false;
	}

	std::cout << "Added a new TextReminder that will go off in " <<
timeAmmount << " " << timeScale << "!" << std::endl;

	return true;
}
#pragma once
#include <iostream>
#include <vector>
/* Basic command format :

The command name is case insensitive as all command names internally and
externally are converted to lowercase.
```

```
Arguments are prefixed by a single dollar sign '$'.
Every section of the command are separated by atleast one space character '
'.

-CommandName $arg0 $arg1 $arg2 $arg3 $arg(n)...

*/

class Command {
public:
	std::string commandTitle;
	std::vector <std::string> commandArgs;
	int commandId;
	Command();
};
#pragma once
#include <iostream>
#include <functional>
#include "Command.h"

class CommandJob {
	std::string commandName;
public:
	CommandJob(const std::string name);

	const std::string& GetCommandName();

	virtual bool OnCommand(const Command& cmd) = 0;
};
#pragma once
#include <iostream>
#include <vector>
#include <map>
#include <memory>
#include "Command.h"
#include "CommandJob.h"
class CommandWork {
	std::string commandTitle;
};

class CommandProcessor {
private:
	std::vector<std::string> rawCommands;
	std::vector<Command> parsedCommands;
	std::map<std::string, std::unique_ptr<CommandJob>> commandJobs; //
Map of key std::string representing the commandname of the job and the
value being the job that the command should execute
	static int commandCount;

public:
	CommandProcessor();

	void AddJobForCommand(std::unique_ptr<CommandJob>& job);


	// Processes an incoming command. If the method fails to parse the
command or the command is unknown, this will return false indicating
failure, else return true
	bool ProcessCommand(std::string rawCmd);

	bool ProcessCommand(const Command& cmd);

};
#pragma once
#include "CommandJob.h"
class PrintValueJob : public CommandJob {
public:
	PrintValueJob();

	virtual bool OnCommand(const Command& cmd);

};
#pragma once
#include <chrono>
#include <iostream>
class Reminder {
	std::chrono::time_point<std::chrono::steady_clock> startTime,
endTime;
	bool running;
public:

	Reminder();

	std::chrono::duration<double> GetTimeLeft();
	void Start(std::chrono::nanoseconds timeTillDone);

	void Stop(bool didWholeTimeElapse);

	bool IsRunning();

	std::chrono::time_point<std::chrono::steady_clock> GetStartTime();

	std::chrono::time_point<std::chrono::steady_clock> GetEndTime();

	std::string
ConvertTimeToString(std::chrono::time_point<std::chrono::steady_clock> t);

	virtual void OnDone(std::chrono::duration<double> timeElapsed) = 0;

	virtual std::string GetDescription() = 0;
};
#pragma once
#include "CommandJob.h"
class ReminderListJob : public CommandJob {
public:
	ReminderListJob();

	virtual bool OnCommand(const Command& cmd);

};
#pragma once
#include <iostream>
#include <vector>
#include <memory>
#include <thread>
#include "Reminder.h"
class ReminderSystem {
	std::vector<std::unique_ptr<Reminder>> reminders;

public:

	ReminderSystem();

	void AddReminder(std::unique_ptr<Reminder>& reminder,
std::chrono::nanoseconds timeTillDone);
```

```
      void Tick();

      void Start();

      std::vector<std::unique_ptr<Reminder>>& GetReminders();
};

extern ReminderSystem* pReminderSystem;
#pragma once
#include "Reminder.h"
#include <iostream>
#include <string>
class SoundReminder : public Reminder {
public:
      SoundReminder();

      virtual void OnDone(std::chrono::duration<double> timeElapsed);

      virtual std::string GetDescription();
};
#pragma once
#include "CommandJob.h"
class SoundReminderJob : public CommandJob {
public:
      SoundReminderJob();

      virtual bool OnCommand(const Command& cmd);

};
#pragma once



#include "Reminder.h"
#include <iostream>
#include <string>
class TextReminder : public Reminder {
      std::string textToShow;
public:
      TextReminder(std::string text);
```

```
	virtual void OnDone(std::chrono::duration<double> timeElapsed);

	virtual std::string GetDescription();
};
#pragma once

#pragma once
#include "CommandJob.h"
class TextReminderJob : public CommandJob {
public:
	TextReminderJob();

	virtual bool OnCommand(const Command& cmd);

};
```