**Реализация подсистемы запуска**

На данном этапе требуется запустить решение и проверить правильность его работы, также в данной подсистеме будут учтены ограничения по памяти и по времени, при превышении величин которых проверка должна закончиться с соответствующим кодом.

В случаях с языками C#, C++ и Pascal успешная компиляция сформирует исполняемый файл \*.exe, а для языка Java будет сформирован бинарный файл \*.class, который впоследствии будет запускаться с помощью Java Virtual Machine.

Запуск будет производиться в отдельном процессе, как и компиляция, только в случае с запуском помимо контроля над временем исполнения (таймаут) будет вестись и контроль над выделенной программе памятью. Для того, чтобы запуск программы производился корректно и в определенной папке необходимо задать будущему процессу параметры, в число которых входят путь к исполняемому файлу и рабочая директория. Назначение этих параметров происходит во фрагменте кода приведенном ниже.

processInfo = new ProcessStartInfo()

{

FileName = Path.Combine(this.path, this.runner),

CreateNoWindow = true,

UseShellExecute = false,

WorkingDirectory = this.path

};

После того, как заданы параметры процесса, его необходимо запустить с одновременным запуском таймера, который будет контролировать время исполнения. Ниже приведен фрагмент кода, где производится запуск процесса и таймера.

Stopwatch sWatch = new Stopwatch();

sWatch.Start();

...

process = Process.Start(processInfo)

...

При успехе запуска процесса сразу же начинается контроль за параметрами запуска: стоит процессу превысить хотя бы один из показателей, его выполнение прерывается и формируется отчетность о проверке решения. Ниже приведен фрагмент кода, в котором осуществляется контроль над выполняемым процессом.

public void StartTestT(){

... /\*задание параметров запуска \*/

//запустить для каждого набора тестов

for (int i = 0; i < this.test\_num; i++){

//перенести набор тестов

MoveTestsToDirectory(i + 1);

// пока процесс выполняется и не превышены показатели осуществляется считывание инф-ии

while (!process.HasExited && !tle && !mle){

process.WaitForExit(50);//периодичность считывания информации 50 мс

process.Refresh();//обновить информацию о процессе

try

{//считываие памяти в кб

if (!process.HasExited) cur\_memory = process.WorkingSet64 / 1024;

}

catch(System.SystemException e){//процесс выполнился

break;

}

//запомнить текущую память

if (cur\_memory > max\_memory) max\_memory = cur\_memory;

//контроль времени выполнения

if (sWatch.ElapsedMilliseconds > this.time\_limit) { tle = true; break; }

//контроль за выделенной памятью

if (max\_memory > this.memory\_limit) { mle = true; break; }

}

sWatch.Stop();//остановка таймера при окончании контроля

if (!process.HasExited)//если процесс все еще продолжается, его необходимо завершить

{

process.Kill();

process.Dispose();

}

/\*Формирование отчетностей в зависимости от результатов запуска \*/

/\*проверка выходных файлов\*/

/\*Формирование отчетностей в зависимости от результатов сравнения файлов \*/

...

}

Итак, считывая память с помощью параметра процесса **WorkingSet64**, который хранит количество выделенной процессу физической памяти в байтах и контролируя время выполнения с помощью таймера, осуществляется контроль за запуском. При этом запоминаются их максимальные значения.

Поскольку проверка проходит на N наборах тестов, то процедуру запуска решения необходимо проводить N раз. При этом каждый раз на вход программе требуется новый входной файл, для переноса содержимого этого файла используется функция **MoveTestsToDirectory(index)**, которая извлекает из папок эталонных тестов i-ый входной файл для проверяемой задачи.

Программа участника должна считать этот файл и на выходе представить выходной файл, которой впоследствии будет сравниваться с эталонным выходным файлом, соответствующим эталонному входному. Сравнение происходит в функции filesAreEqual(FileInfo1, FileInfo2). Ее исходный код приведен ниже.

private bool filesAreEqual(FileInfo first, FileInfo second)

{

using (StreamReader sr1 = first.OpenText(), sr2 = second.OpenText())

{

string s1 = "", s2 = "";

s1 = sr1.ReadLine();

s2 = sr2.ReadLine();

while (s1 != null && s2 != null)

{

if (s1 != s2) return false;

s1 = sr1.ReadLine();

s2 = sr2.ReadLine();

}

if (s1 == s2 || (s1 == null && (s2 == "\n" || s2 == "")) || (s2 == null && (s1 == "\n" || s1 == ""))) return true;

}

return false;

}

Можно было произвести более быстрое побайтовое сравнение файлов, однако в случае с построчным сравнением предусмотрено наличие/отсутствие перехода на новую строку в конце обоих файлах. Это случай, при котором все значимые числа/символы выходного файла участника совпадают со всеми значимыми числами/символами эталонного, то есть ответ абсолютно верный. Однако в конце файла участника не стоит перехода на новую строку, в то время как в эталонном он есть, поэтому это небольшое различие может привести к появлению ошибки. Байтовый просмотр файла не учитывает такую возможность, и в случае ее появления выдаст признак несовпадения файлов.

Ниже приведены участки кода, ответственные за выдачу кодов проверки в зависимости от результатов запуска.

TLE:

Если на этапе контроля запуска решения лимит времени был превышен, то переменная tle будет равна true, и сработает нижестоящий фрагмент кода, будет сформирована отчетность и проверка прекратится.

if (tle){//флаг tle устанавливается при контроле запуска

REPORT.TLE = true;

REPORT.Message = "Time limit exceeded.";

REPORT.finished=true;

... /\*запись в логи \*/  
}

MLE:

Срабатывает аналогично TLE, только при контроле за памятью.

else if (mle){ //флаг mle устанавливается при контроле запуска

REPORT.MLE = true;

REPORT.Message = "Memory limit exceeded.";

REPORT.finished=true;

... /\*запись в логи \*/  
}

RE:

Любое решение, если оно выполнилось успешно должно вернуть нулевой код возврата. Если такой код не возвращается, то скорее всего в программе есть необработанные исключения.

if (process.HasExited && process.ExitCode != 0)

{//программа завершилась вовремя, но вернула не 0.

REPORT.RE = true;

REPORT.Message = "Runtime error.";

REPORT.finished = true;

... /\*запись в логи \*/

}

PE:

Данная ошибка возникает, когда вывод производился в файл с ошибочным именем или вовсе не был осуществлен. Определяется путем проверки существования выходного файла, который дожна была сформировать программа участника

if (!File.Exists(Path.Combine(this.path, this.output\_filename)))

{

REPORT.PE = true;

REPORT.Message = "No output file.";

REPORT.finished = true;

... /\*запись в логи \*/

}

WA:

Срабатывает в случае, если сравнение файлов дает отрицательный результат.

if (!this.CompareOutputFiles(i + 1))

{

REPORT.WA = true;

REPORT.Message = "Wrong answer.";

REPORT.finished = true;

... /\*запись в логи \*/

}

OK:

Если ни одна из вышеперечисленных ветвей кода не была выполнена ни на одном наборе из тестов, то решению выдается код проверки OK, то есть решение прошло все стадии проверки. Поэтому в отличии от всех остальных ветвей проверки в данном случае требуется записать показатели памяти и времени, поскольку они будут выступать в роли главных параметров оценивания в будущем.

REPORT.OK = true;

REPORT.memory = max\_memory;

REPORT.time = max\_time;

REPORT.Message = "Solution accepted.";

REPORT.finished = true;

По прохождении любой из ветвей перечисленных выше формируется отчетность, которая передается в функцию, вызвавшую систему проверки. Такой функцией является функция запуска решения из очереди, которая впоследствии и сохраняет записанные в отчетность результаты в таблицу.

**Итог**

Любое решение, которое направлено в систему проверки, должно пройти через три стадии: проверка запрещенных символов, компиляция и запуск. Попадание на последующую стадию возможно только при условии того, что пройдены предыдущие. В случае непрохождения любой из стадий формируется отчетность с кодом проверки, в случае же прохождения всех трех стадий в отчетности будет указан код проверки OK и записаны параметры запуска.

Контроль над ходом компиляции и запуска осуществляется с помощью создания процесса и мониторинга затрачиваемых им ресурсов с определенной периодичностью. При превышении времени работы этого процесса он будет завершен и будет сформирована соответствующая отчетность о прохождении проверки.

Все три подсистемы имеют строго разделенные обязанности, поэтому внесение изменений в одну из них не повлияет на функционирование остальных.

**Реализация системы оценивания**

Заключительным пунктом любого олимпиадного решения является система оценивания. Чтобы дать участникам какой-либо олимпиады возможность просматривать места, которые они заняли, система оценивания должна сформировать рейтинг участников. В таком рейтинге пользователи должны располагаться в порядке убывания общего количества очков, которое подсчитывается на основе решенных ими задач.

Началом работы системы оценивания было принято считать момент получения данных из таблиц результатов.

**Получение и обработка данных**

Для начала работы системы оценивания для определенной олимпиады необходим следующий набор данных:

* Список участников
* Список задач
* Список результатов

Сделав определенные запросы к таблицам «Участники», «Олимпиадные задачи» и «Решения олимпиадных задач» соответственно, можно получить все три списка.

Контроллер веб-приложения получает идентификатор олимпиады, для которой необходимо сформировать рейтинг. Таким образом, из всех трех таблиц необходимо выбрать лишь те записи, в которых фигурирует данный идентификатор олимпиады.

С помощью сгенерированного реляционным конструктором объектов контекстом [**MainDB]DataContext** (где **MainDB** – имя базы данных) устанавливается соединение веб-приложения с базой данных. А с помощью сгенерированных сущностных классов можно выполнять запросы к базе данных.

В первом случае сущностным классом будет являться участник, а по названию таблицы «olimpiad\_party». Поэтому, чтобы получить список участников какой-либо олимпиады необходимо обратиться к функции вида:

static public List<Olimpiad\_party> GetOlympParticipantsById(int o\_id)

{

var context = new MainDBDataContext();

List<Olimpiad\_party> R= context.Olimpiad\_party.Where(s => s.olymp\_id == o\_id).ToList();

context.Dispose();

return R;

}

Из всех частников олимпиады будут выбираться лишь идентификаторы тех, кто записан на выбранную олимпиаду.

Подобным образом извлекаются все остальные данные из таблиц. Однако перед отправкой данных системе оценивания их необходимо определенным образом обработать, поскольку система оценивания принимает входные данные в виде списков специально описанных объектов:

* Список объектов класса **Ranger.Participant**

List<Diplom.Ranger.Participant> Participants = new List<Diplom.Ranger.Participant>();

* Список объектов класса **Ranger.TaskInfo**

List<Diplom.Ranger.TaskInfo> TasksInfo = new List<Diplom.Ranger.TaskInfo>();

* Список объектов класса **Ranger.Result**

List<Diplom.Ranger.Result> Results = new List<Diplom.Ranger.Result>();

Вышеуказанные классы практически полностью копируют набор полей из таблиц «Пользователь», «Задача» и «Решение олимпиадной задачи». Ниже в качестве примера приведено определение одного из таких классов.

public class TaskInfo

{

public int id\_t { get; set; }

public long time\_limit { get; set; }

public long memory\_limit { get; set; }

public float points { get; set; }

public TaskInfo(int \_id\_t, long \_tl, long \_ml, float \_p)

{ /\*инициализация\*/}

}

Поскольку из таблиц были извлечены лишь идентификаторы пользователей и идентификаторы задач, то по этим идентификаторам необходимо получить наборы параметров соответствующих полям того или иного класса. Так, например, извлекаются нужные параметры для объекта класса **TaskInfo**.

Class PagesController{

...

[HttpPost]

public ActionResult WriteReport(string olimp)

{

...

foreach(Olimpiad\_tasks TI in \_TasksInfo){

Tasks T=Diplom.WorkWithBD.GetTaskById(TI.Id\_task);

TasksInfo.Add(new Diplom.Ranger.TaskInfo(TI.Id\_task,(long)T.time\_limit,(long)T.memory\_limit, (float)T.points));

}

.../\*отправка данных в систему оценивания\*/

}

...

}

В конце, когда сформированы все наборы данных, они отправляются в систему оценивания, которая, так же как и система проверки, по окончании формирует отчетность.

Diplom.Ranger.Report r=Diplom.Ranger.RangerProgram.RUN(TasksInfo,Participants,Results, id\_o);

**Отчетность системы оценивания**

Если представить получаемую отчетность в виде таблицы, то она выглядит следующим образом:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Место | Участник | Задача #1 | Задача #45 | … | Задача #745 | Итоговое количество баллов |
| 1 | Участник #45 | Код: OK  Память: 2856 кб  Время: 100 мс  Попытки: 1  Баллы: 10.0 | Код: OK  Память: 65423 кб  Время: 1140 мс  Попытки: 2  Баллы: 7.8 | … | Код: OK  Память: 9652 кб  Время: 652 мс  Попытки: 1  Баллы: 5.0 | 98.4 |
| 2 | Участник #2 | Код: OK  Память: 3567 кб  Время: 123 мс  Попытки: 1  Баллы: 8.0 | Код: OK  Память: 62451 кб  Время: 1100 мс  Попытки: 1  Баллы: 8.9 | … | Код: OK  Память: 8000 кб  Время: 620 мс  Попытки: 1  Баллы: 6.0 | 96.2 |
| … | … | … | … | … | … | … |
| 541 | Участник #6 | Код: CE  Память: - кб  Время: - мс  Попытки: 5  Баллы: 0.0 | Код: RE  Память: - кб  Время: - мс  Попытки: 7  Баллы: 0.0 | … | Код: SV  Память: - кб  Время: - мс  Попытки: 3  Баллы: 0.0 | 0.0 |

В данной таблице содержится информация обо всех участниках, задачах, всех результатах решения и итоговых количествах баллов, по которым в левой колонке выставлены места.

При перенос данной таблицы в программный вариант, можно представить каждую строку как запись, которая содержит информацию об участнике, его место, количество очков и задач, которые он решал/не решал. Такую таблицу в программном варианте представляет класс **ReportNote.**

public class Report

{

public class ReportNote

{

public class ReportTask{

public int id\_t;//идентификатор задачи

public string result;//код проверки

public int attempts;//количество попыток

public long memory;//затраченная память

public long time;//затраченное время

public float points;//количество очков за задачу

}

public int id\_p;//идентификатор участника

public float total\_points;//итоговое количеств очков

public int place;//место

public List<ReportTask> Tasks;//записи о результатах проверки решения

}

...

public List<ReportNote> ReportNotes;

}

В такую отчетность должен быть записан итоговый результат ранжирования.

Также класс **Report** содержит список всей информации по олимпиаде: списки участников, задач и решений. Эти списки в дальнейшем будут использоваться во время ранжирования.

public List<TaskInfo> TasksInfo;

public List<Participant> Participants;

public List<Note> Participant\_Notes;

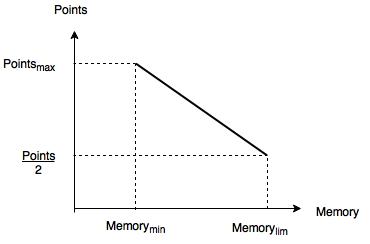
Последний список представляет собой структуру аналогично устроенную точно так же как и строки из таблицы \*\*\*, то есть определение этого класса практически совпадает с определением класса **ReportNote**. Различие между ними в основном функциональные, объекты класса **ReportNote** используются лишь для вывода информации на страницу, а объекты класса **Note** имеют функционал для обработки решения, подсчета очков и т.д.

**Ранжирование результатов**

Как было сказано в главе посвященной проектированию системы оценивания, на балл влияют три параметра: затраты памяти, затраты времени и количество попыток. Чем сильнее показатель памяти или времени отдален от лучшего и приближен к лимитирующему, тем меньший балл получит решение. При этом разделены получаемые очки между показателями времени и памяти поровну, то есть если за задачу дается десять очков, то максимум за временной показатель можно получить пять, как и за показатель памяти. Каждая попытка отнимает от итогового балла по одному очку.

При проставлении результатов за каждую задачу учитываются значения Memorymin(минимальная затраченная память на решение определенной задачи, для каждой задачи это значение различно), Memorylim (лимитирующее значение памяти, достижение которого даст решению не больше половины очков), Timemin (минимальное затраченное время на решение определенной задачи), Timelim (лимтирующий показатель времени).

Зависимость выставляемых очков можно показать графике, изображенном на рисунке \*\*\*.



В качестве графика, представляющего зависимость выставления балла от показателя (памяти или времени), может быть задана и кривая, например вогнутая гипербола, на начальных значениях которой выставляемые очки будут снижать очень быстро, а после это падение замедлится.

Функция, которая линейно определяет выставляемое количество очков, при известных показателях Memorymin/ Memorylim или Timemin/ Timelim в зависимости от полученного значения показателя, выглядит следующим образом:

private float LinearFunc(float x1, float x2, float y1, float y2, float param)

{

if (x2 == x1) return y1;

return (param \* (y2 - y1) - x1 \* (y2 - y1)) / (x2 - x1) + y1;

}

Указанная выше функция используется при проставлении баллов за задачу. Для того чтобы использовать эту функцию необходимо заранее знать Memorymin/ Memorylim или Timemin/ Timelim.

Лимитирующие показатели известны из условия задачи. Лучшие же показатели требуется выбрать среди множества показателей решений участников.

Для этой цели используется выборка того результата из всех в котором эти параметры минимальны. Приведенный ниже код выполняет эту операцию:

long min\_time = ResultsForTask.Where(result => result.result == "OK").ToList().Min(res => res.time);

long min\_memory = ResultsForTask.Where(result => result.result == "OK").ToList().Min(result => result.memory);

Из списка решений для текущей задачи выбираются лучшие показатели и сохраняются в переменных, которые в дальнейшем будут участвовать при выставлениях баллов.

Выставление баллов происходит в функции **Range()** класса **Ranger.** В данной функции происходит обработка объектов из списка результатов олимпиады **Results**. Для каждой задачи из перечня олимпиадных задач (для определенной олимпиады) выявляются лучшие показатели и на их основе подсчитываются остальные с помощью линейной функции, описанной выше. При этом оценивание производится только тех результатов, код проверки которых OK, всем остальным решениям по умолчанию выставлено 0 баллов.

class Ranger{

...

public void Range()

{

...

foreach (TaskInfo TI in TasksInfo)

{

List<Result> ResultsForTask = Results.Where(result => result.id\_t == TI.id\_t).ToList();

if (ResultsForTask.Count != 0)

{

long min\_time = ResultsForTask.Where(result => result.result == "OK").ToList().Min(res => res.time);

long min\_memory = ResultsForTask.Where(result => result.result == "OK").ToList().Min(result => result.memory);

long time\_limit = TI.time\_limit;

long memory\_limit = TI.memory\_limit;

//лучший по времени получает максимум, и чем ближе к лимиту, тем ближе будет оценка к половине балла

float min\_points = 0.5f;

float max\_points = 1.0f;

foreach (Result R in ResultsForTask)

{

//объект класса Points, который хранит выставленные задаче очки, по умолчанию равны 0

Points p = new Points(TI.points, turn\_PP);

if (R.result == "OK")//выставление очков

{

p.AP = R.attempts;//попытки

p.TP = LinearFunc(min\_time, time\_limit, max\_points, min\_points, R.time);//очки за время

//очки за память

p.MP = LinearFunc(min\_memory, memory\_limit, max\_points, min\_points, R.memory);

}

//сохранить информацию об очках в записях участников

REPORT.SetParticipantTaskInfo(R.id\_p, R.id\_t, p, R.result, R.attempts, R.time, R.memory);

}

...

}

}

...

}

Количество очков для каждой задачи задается с помощью объекта класса **Points**, его определение представлено ниже.

public class Points

{

public float OP { get; set; }//максимальное количество очков [1-N]

public float AP { get; set; }//штраф за количество попыток [0-M]

public bool turn\_PP { get; set; }//учитывать ли параметры запуска при выставлении баллов

public float MP { get; set; }//количество очков за результат выполнения по параметру память [0.5-1]

public float TP { get; set; }//количество очков за результат выполнения по параметру время [0.5-1]

...

/\*конструкторы\*/

public float getPoints()

{

if (AP == 0) return 0;

float points = (MP + TP) \* OP / 2 - (AP - 1);

if (points <= 0) return 0;

return points;

}

}

В данном классе фигурирует формула, ответственная за подсчет очков для данной задачи. Данная формула означает:

Где *RP* – количество баллов за задачу, *MP* – количество очков за показатель памяти, *TP* – количество очков за показатель времени, *AP* – количество попыток.

В программе MP и TP принимают значения 0 и [0.5, 1.0] в зависимости кода проверки и от показателей решения студента.

На данном этапе все участники олимпиады получили очки за все свои решения, основываясь на результатах остальных. Но список участников не отсортирован, и никому из них пока не назначены места.

Функция класса **Ranger Sort()** сортирует записи об участниках, основываясь на общем количестве очков каждого. Ее код приведен ниже.

public void Sort()

{

//сортировка записей

REPORT.Participant\_Notes.Sort(delegate(Note x, Note y)

{

return (y.TotalPoints().CompareTo(x.TotalPoints())); //по убыванию

});

// проставление мест

if (REPORT.Participant\_Notes.Count != 0)

{

float prev\_result = REPORT.Participant\_Notes[0].TotalPoints();

int place = 1;

foreach (Note N in REPORT.Participant\_Notes)

{

if (prev\_result > N.TotalPoints()) place++;

N.place = place;

prev\_result = N.TotalPoints();

}

}

Сортировка производится с помощью делегата, сравнивающего общие очки участников. Записи участников располагаются по убыванию.

На этом процедура ранжирования окончена, остается лишь сформировать отчетность. Она формируется в двух видах: в виде файла, где информация заключена между тегами, и в виде списка объектов, который пересылается в контроллер, где впоследствии его содержимое выводится на страницу.

При этом намного эффективнее при дальнейших запросах будет использовать именно файл, чем вновь производить операции описанные выше.

**Итог**

Заключительный этап обработки решения олимпиадной задачи, этап оценивания, успешно реализован. Дробность оценки и маловероятно совпадающие между собой показатели, использующиеся при оценивании, минимизируют количество разделяемых мест и придают оценке объективность. Поскольку сравнение по параметрам запуска является наиболее объективным при решениях задач по программированию.

В будущем к этим параметрам легко можно добавить другие критерии, например критерий, оценивающий длину решения. Достаточно будет добавить в формулу подсчета данный коэффициент, и решение уже будет оцениваться по четырем параметрам.