УДК 517.977.5
ББК в.6.3.1.0

ПОДСИСТЕМА «РАЗРАБОТЧИК» СИСТЕМЫ ПРИЕМА КОММУНАЛЬНЫХ ПЛАТЕЖЕЙ

Клименко А. А.[1], Угольницкий Г.А.[2]
(*Южный федеральный университет, Ростов-на-Дону*)

В работе рассматривается одна из ключевых подсистем приема коммунальных платежей – «Разработчик». Описана разработанная система массового обслуживания, которая моделирует данную подсистему. Поставлена и решена задача о распределении ресурсов (в решении использовался модифицированный «венгерский» алгоритм). В заключительной части статьи приведено описание имитационной (агентной) модели данной подсистемы и результаты имитационных экспериментов.

Ключевые слова: система массового обслуживания, задача о распределении, модификация «венгерского» алгоритма, имитационное моделирование, агентная модель, имитационный эксперимент

1. Введение

Для системы приема коммунальных платежей наиболее важным является вопрос распределения ресурсов во времени. Под ресурсами понимается как обслуживающий персонал, так и аппаратно-программный комплекс, на базе которого работает система. С точки зрения дискретно-событийного моделирования данная система представляет собой систему массового

[1] *Клименко Антон Александрович*, *магистр прикладной математики и информатики* (*antklim@gmail.com*).
[2] *Угольницкий Геннадий Анатольевич, заведующий кафедрой прикладной математики и программирования, д.ф.-м.н., профессор (ougoln@mail.ru)*

обслуживания с разнородным потоком заявок и таким же разнородным множеством механизмов, обрабатывающих эти заявки. Из-за такой гетерогенности системы дискретно-событийная модель данной системы не способна описать ее полностью. Для этого необходимо использование методов имитационного моделирования.

2. Анализ системы приема коммунальных платежей

При детальном рассмотрении общей системы приема и обработки коммунальных платежей в ее составе можно выделить подсистемы массового обслуживания, описывающие работу следующих процессов:

1. процесс приема платежа;
2. процесс обработки информации о платежах (например, процессы порождения платежных документов в централизованной базе данных платежей);
3. процесс поддержки пользователей при работе прикладным программным обеспечением;
4. процесс разработки дополнительных модулей прикладной программы и доработки существующих с целью оптимизации их работы.

Данные подсистемы соответственно можно назвать «Кассир», «Центральная База Данных», «Администратор» и «Разработчик». В данной статье подробно рассмотрим одну из подсистем, а именно подсистему «Разработчик».

3. Система массового обслуживания «Разработчик»

Как правило, над системой работает от одного до N разработчиков. Задания, поступающие к разработчикам, характеризуются по нескольким критериям. Первый из них – тип задачи. Можно выделить три типа задач: исправление ошибок в существующем ПО, разработка новой функциональности, изменение существующей

функциональности. В зависимости от типа задачи определяется приоритет ее выполнения.

С другой стороны, поступающие задачи различаются по степени их сложности либо объему работ. Объединение этих двух факторов обосновано тем, что единицей измерения работ являются человеко-часы, потраченные на задачу.

На основе вышеизложенного определим основные характеристики системы массового обслуживания, а затем определим, к какому типу систем она относится. Очевидно, что входным потоком здесь является поток заявок на разработку/доработку ПО. Обслуживающим прибором здесь выступает разработчик. Выходным потоком являются функционирующие части ПО, которые подверглись изменениям. Пусть D – множество приборов, $D \neq 0$, $D = \{d_i, i = 1, \ldots, n\}$. Каждый разработчик имеет очередь бесконечной длины. Таким образом, возникает сеть с N узлами.

Далее рассмотрим природу входного потока системы. Так как существует несколько типов задач, то имеем дело с **неординарным** входным потоком. Пусть в системе имеется $j = 1, \ldots, H$ типов заявок. Так как интервалы времени между соседними заявками любого из j-го класса являются случайными величинами, то получаем H случайных потоков. Пусть λ_{ij} – интенсивность потока заявок j-го типа в i-м узле системы. Так как λ_{ij} не меняется во времени, то имеем дело со **стационарным** потоком. Эта система относится к системам **без последействия**, так как заявки поступают независимо друг от друга и момент поступления очередной заявки не зависит от поступления предыдущих заявок. По классификации дисциплин буферизации заявок во входном потоке, данный поток относится к **бесприоритетным** потокам. С точки зрения классификации дисциплин обслуживания имеем **приоритетную систему одиночного типа**, так как всякий раз на обслуживание назначается только одна задача. Следовательно, согласно обозначениям Кендалла получаем систему $\overline{M_H} / \overline{G_H} / N$. Характеристики по каждому классу заявок $j = 1, \ldots, H$

идентичны СМО с однородным потоком. Суммарный поток заявок характеризуется такими показателями, как

Суммарная интенсивность потока:

(1) $\Lambda = \sum_{i=1}^{N} \sum_{j=1}^{H} \lambda_{ij}$.

Суммарная нагрузка и суммарная загрузка:

(2) $Y = \sum_{i=1}^{N} \sum_{j=1}^{H} \rho_{ij}$.

(3) $R = \min(\sum_{i=1}^{N} \sum_{j=1}^{H} r_{ij}, 1)$.

где ρ_{ij} – нагрузка, создаваемая заявками класса j в узле i.

$r_{ij} = \min(\frac{\rho_{ij}}{N}; 1)$.

Условие отсутствия перегрузок в такой СМО имеет вид R < 1.

Среднее время ожидания W и среднее время пребывания U заявок объединенного потока в системе соответственно равны:

(4) $W = \sum_{i=1}^{N} \sum_{j=1}^{H} \alpha_{ij} \omega_{ij}$.

(5) $U = \sum_{i=1}^{N} \sum_{j=1}^{H} \alpha_{ij} \upsilon_{ij}$.

где $\alpha_{ij} = \lambda_{ij} / \Lambda$ – коэффициент, учитывающий долю заявок класса j для i-го узла в суммарном потоке, ω_{ij} – время ожидания заявок в очереди, υ_{ij} – время пребывания заявок в системе.

Суммарная длина очереди и суммарное число заявок в сети определяются следующими соотношениями:

(6) $L = \sum_{i=1}^{N} \sum_{j=1}^{H} l_{ij}$.

(7) $M = \sum_{i=1}^{N} \sum_{j=1}^{H} m_{ij}$.

где l_{ij} – длина очереди заявок, m_{ij} – число заявок в системе. Из полученных соотношений легко можно получить следующие выражения:

(8) $U = W + B, L = \Lambda W, M = \Lambda U$.

Здесь B – среднее время обслуживания любой заявки суммарного потока, которое вычисляется по следующей формуле:

$$(9) \quad B = \sum_{i=1}^{N} \sum_{j=1}^{H} \alpha_{ij} b_{ij} \ .$$

где b_{ij} – среднее время обслуживания заявки *j*-го типа в *i*-м узле.

Итак, определено, к какому типу систем массового обслуживания относится данная система, и выделены ее основные характеристики. Далее перейдем к постановке оптимизационной задачи для данной системы.

4. Оптимизация системы массового обслуживания «Разработчик»

Задачу оптимизации для данной системы можно сформулировать следующим образом – уменьшить время простоя разработчика, уменьшить время доработки. Однако природа входного потока в данной системе такова, что здесь возможно применение теории сетевого планирования и управления проектами. Ввиду использования в процессе разработки методологии экстремального программирования[1] долгосрочное планирование зачастую оказывается бесполезным. В данном случае наиболее подходящим является формулировка задачи о назначениях. Сформулируем данную задачу.

Пусть x_1 , x_2 , …, x_n – разработчики, t_1 , t_2 , …, t_n – типы заданий, которые выполняются разработчиками, $f(x_i , t_j)$ – время выполнения *i*-м разработчиком *j*-го задания, $i = 1, \ldots, n; \ j = 1, \ldots, m$. Тогда необходимо найти допустимое назначение, у

---

[1] *Экстремальное программирование (Extreme Programming, XP) – дисциплина разработки программного обеспечения и ведения бизнеса в области создания программных продуктов, которая фокусирует усилия обеих сторон (программистов и бизнесменов) на общих целях[3].*

которого суммарное время выполнения работ будет минимальным. Наиболее эффективным методом решения данной задачи является «венгерский» алгоритм [5]. Далее приведем решение поставленной задачи с использованием модифицированного «венгерского» алгоритма. Для начала рассмотрим простой случай с ***n*** разработчиками, ***m*** задач и значениями ***n*** = ***m***. Построим исходную матрицу стоимостей работ:

Таблица 1. Исходная матрица стоимости работ

	1	2	3	…	m
1	C_{11}	C_{12}	C_{13}	$C_{1...}$	C_{1m}
2	C_{21}	C_{22}	C_{23}	$C_{2...}$	C_{2m}
…	$C_{...1}$	$C_{...2}$	$C_{...3}$		$C_{...m}$
n	C_{n1}	C_{n2}	C_{n3}	$C_{n...}$	C_{nm}

где C_{ij} , $i = 1, ..., n;$ $j = 1, ..., m$ – время выполнения i-м сотрудником j-го типа задания.

Так как в качестве решения необходимо знать, какой разработчик должен выполнять ту или иную работу, то решение можно записать в виде массива D:

(10) $D = \{d_1 ,d_2 ...,d_n\}$

Здесь индекс d соответствует номеру работы, а значение d_j соответствует номеру разработчика, назначенного на j-ю работу. На каждом шаге алгоритма будем выбирать для каждой строки минимальное значение. Затем выберем минимальное из этих минимальных. Если минимальные значения совпадают, то берем значения, находящиеся в северо-западном углу (не всегда верно для матриц 2*2) матрицы. Записываем номер строки минимального значения в соответствующую ячейку массива D. Удаляем строку и столбец, на пересечении которых находится найденный элемент. Повторяем шаг с новой матрицей.

Преимущество данного алгоритма над стандартным «венгерским» заключается в том, что на каждом шаге вычислительная сложность уменьшается, так как уменьшается размерность исходной матрицы, соответственно уменьшается количество элементов, подверженных обработке. Однако и данный алгоритм можно оптимизировать.

На первом шаге алгоритма для каждой строки матрицы можно выбрать минимальное значение и номер столбца, в котором находится выбранный элемент (связано с тем, что на каждом шаге алгоритма значения элементов матрицы не меняются). Учтем замечания о выборе минимального элемента для матриц размерности 2*2.

Шаг 1.

(11) $D = \{\underbrace{0, 0, \dots, 0}_{m}\}$

(12) $C = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1m} \\ c_{21} & c_{22} & \cdots & c_{2m} \\ & \cdots & & \\ c_{n1} & c_{n2} & \cdots & c_{nm} \end{pmatrix} \rightarrow M = \{M_1, M_2, \dots, M_n\}$

(13) $M_i = \begin{cases} \min(c_{i1}, c_{i2}, \dots, c_{im}) = \overline{m} \\ J = \{j : c_{ij} = \overline{m}\} \end{cases}$

Каждый M_i содержит в себе минимальное значение и индексы столбцов, в которых они встречаются. Дальнейшая работа ведется с массивом M. Чтобы не зависеть от порядка следования элементов в массиве M и иметь возможность удалять элементы из массива, модифицируем структуру элемента массива (13). Добавим номер строки.

(14) $M_i = \begin{cases} i \\ \min(c_{i1}, c_{i2}, \dots, c_{im}) = \overline{m} \\ J = \{j : c_{ij} = \overline{m}\} \end{cases}$

Шаг 2.

Находим M_i с наименьшим $\overline{m}$ и наименьшим значением i. Если элементов в массиве дольше двух, заносим i в массив D в ячейку с номером, совпадающим с наименьшим значением j M_i-го элемента. Затем M_i можно удалить из массива M, а все элементы с таким же значением j заменить на элемент с минимальным значением, полученный из столбцов матрицы с другими номерами, и вернуться на начало шага 2.

На последней итерации, когда в массиве M останется два элемента, необходимо сделать проверку на пересечение значений j в оставшихся элементах, и выбрать непересекающиеся.

Теперь рассмотрим случай $\boldsymbol{n > m}$. Шаги 1 и 2 будут такими же, как и для равных значений. Отличие только в финальном шаге. На последней итерации в массиве M останется $\boldsymbol{n - m + 2}$ элемента. Основываясь на данных из предыдущих итераций об обработанных i и j, получаем итоговую матрицу размерности $\boldsymbol{(n - m + 2)*2}$. Для каждой строки матрицы получим сумму первого элемента со вторыми элементами остальных строк. Индексы строк элементов, давших наименьшую сумму, заносятся в соответствующие позиции итогового вектора D.

В случае $\boldsymbol{n > m}$ алгоритм будет применен $\frac{m}{n} + 1$ раз. Первый шаг такой же, как и ранее. На втором шаге вычеркиваем самый левый столбец матрицы, в котором содержится наименьший элемент. Строку, в которой находился этот элемент, помечаем временно использованной. Далее выбираем минимальный элемент по оставшимся строкам матрицы. Возвращаемся на начало шага 2 и удаляем очередной столбец матрицы. Как только возникнет ситуация, когда останется только одна непомеченная как используемая строка матрицы, то выбираем в ней минимальный элемент и удаляем из матрицы крайний левый столбец, содержащий данный элемент. После этого снимаем пометки на строках и возвращаемся к началу шага 2. Данная итерация продолжается, пока не будут обработаны все столбцы матрицы.

Аналогично первым двум случаям исходную матрицу можно представить в виде массива.

$$(15)\ C = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1m} \\ c_{21} & c_{22} & \cdots & c_{2m} \\ & \cdots & & \\ c_{n1} & c_{n2} & \cdots & c_{nm} \end{pmatrix} \rightarrow M = \{M_1, M_2, \ldots, M_n\}$$

В отличие от рассмотренных ранее алгоритмов структура элемента M_i будет иной.

$$(16)\ M_i = \begin{cases} i \\ u = \overline{0,1} \\ \min(c_{i1}, c_{i2}, \ldots, c_{im}) = \overline{m} \\ J = \{j : c_{ij} = \overline{m}\} \end{cases}$$

Здесь параметр *u* принимает значение 0, если элемент M_i разрешено использовать, и 1, если M_i временно использован.

Шаг 2 данного алгоритма отличается от рассмотренных для случаев ***n = m, n > m*** тем, что M_i элемент не удаляется из массива, а помечается как временно использованный. Данная пометка снимается со всех элементов, как только не найдено более свободных.

Таким образом, был описан алгоритм для разового нахождения оптимального назначения. Однако более реалистичной является ситуация, когда разработку ведет группа специалистов и сама задача делится на подзадачи. Необходимо подобрать группу специалистов для максимально быстрого решения общей задачи. Иными словами, необходимо последовательно решить ряд задач о назначении и при этом получить оптимальную последовательность.

Сначала рассмотрим случай, когда части основной задачи не зависят друг от друга и могут быть обработаны в любом порядке. Пусть имеется N разработчиков и M типов задач, и основная задача распадается на K подзадач.

$$(17)\ T = \{C_k, k = \overline{1,K}\}$$

где C_k представляют собой матрицы назначения.

Рассмотрим ситуацию N = M = K = 3. Тогда получаем три матрицы.

(18) $$C_1 = \begin{pmatrix} c_{11}^1 & c_{12}^1 & c_{13}^1 \\ c_{21}^1 & c_{22}^1 & c_{23}^1 \\ c_{31}^1 & c_{32}^1 & c_{33}^1 \end{pmatrix} C_2 = \begin{pmatrix} c_{11}^2 & c_{12}^2 & c_{13}^2 \\ c_{21}^2 & c_{22}^2 & c_{23}^2 \\ c_{31}^2 & c_{32}^2 & c_{33}^2 \end{pmatrix} C_3 = \begin{pmatrix} c_{11}^3 & c_{12}^3 & c_{13}^3 \\ c_{21}^3 & c_{22}^3 & c_{23}^3 \\ c_{31}^3 & c_{32}^3 & c_{33}^3 \end{pmatrix}$$

Применив первый шаг алгоритма, получим три массива, для каждой из матриц.

(19) $M^1 = \{M_1^1, M_2^1, M_3^1\}\ M^2 = \{M_1^2, M_2^2, M_3^2\}\ M^3 = \{M_1^3, M_2^3, M_3^3\}$

Применяя алгоритм, получаем три массива решений.

(20) $D^1 = \{d_1^1, d_2^1, d_3^1\}\ D^2 = \{d_1^2, d_2^2, d_3^2\}\ D^3 = \{d_1^3, d_2^3, d_3^3\}$

Полученные результаты можно записать в виде матрицы. Такая запись даст возможность управлять последовательностью работ путем перестановок отдельных строк матрицы.

Таким образом, получено решение задачи о назначениях для задач сложного типа без зависимостей между подзадачами. Был рассмотрен случай для трех подзадач, но данный алгоритм действителен и для любого целого положительного K.

Далее усложним задачу и введем зависимости между подзадачами. Для описания последовательности работ построим орграф, вершинами которого являются подзадачи.

(21) $G = \{g_k, k = \overline{1, K}\}$.

Пусть каждая вершина имеет вес, равный суммарному времени выполнения работ по подзадаче. Тогда решением задачи о назначениях с зависимыми подзадачами станет нахождение пути проходящего через все вершины графа. Суммарное время выполнения работ – сумма весов вершин графа.

Итак, описана задача оптимизации, возникающая в подсистеме «разработчик», и приведен метод ее решения.

Однако данный метод позволяет решать задачу статически, то есть ничего не известно о поведении данной системы во времени. Еще одним недостатком данного метода является невозможность определить взаимосвязь таких важных факторов, как время выполнения работ, сложность работы и стоимость исполнения работы. Добавление всех этих

параметров к рассмотренной задаче увеличит в разы ее размерность, следовательно, и время расчетов. Время расчетов может быть настолько велико, что не сможет уложиться в приемлемые рамки.

Решение такой задачи требует применения совершенно иных методов.

5. Имитационное моделирование подсистемы «Разработчик»

Одним из таких методов является имитационное моделирование, известными достоинствами которого являются гибкость и возможность моделирования систем на любом уровне детализации или абстракции. Первоначально имитационное моделирование определялось как дискретно-событийный способ описания и моделирования процессов, присущих системам массового обслуживания, системам с отказами и восстановлением элементов, дискретными производствами и т.д. [8] .

Наиболее подходящим инструментом для решения поставленной ранее задачи (распределение ресурсов во времени, описание связей различных факторов системы), является агентное моделирование[1].

Агентное моделирование является инструментом, при помощи которого возможно успешное моделирование сложных адаптивных систем. Агентное моделирование базируется на идее моделирования процессов «сверху-вниз»: в основе модели лежит набор основных элементов, из взаимодействия которых

---

[1] *Агентное моделирование (agent-based modeling (ABM)) – метод имитационного моделирования, исследующий поведение децентрализованных агентов и то, как такое поведение определяет поведение всей системы в целом. При моделировании определяется поведение агентов на индивидуальном уровне, а глобальное поведение возникает как результат деятельности множества агентов.*

рождается обобщенное поведение системы. Агентами могут быть не только индивидуумы, но и группы индивидуумов, подсистемы и т.д. Применение данного подхода к моделированию наиболее удобно в случаях, когда представляют интерес характеристики поведения системы в целом, которые определяются как интегральные характеристики всей совокупности агентов [4].

Наиболее распространенным способом описания поведения агентов являются карты состояний (statecharts), которые представляют собой конечный автомат с некоторыми дополнениями. Они описывают переходы между состояниями и события, инициирующие данные переходы, временные задержки и действия, совершаемые агентом на протяжении своей жизни. Агент может иметь несколько параллельно активных и взаимодействующих карт состояний, каждая из которых описывает какой-либо аспект его жизни.

Для рассматриваемой подсистемы можно выделить следующих агентов: заявки на доработки, разработчики.

Простая схема работы заключается в том, что заявки попадают в систему, обслуживаются, задерживаются и покидают систему. Событие генерации очередной заявки будет соответствовать созданию нового агента. На рис. 1 изображена схема жизненного цикла агента заявки.

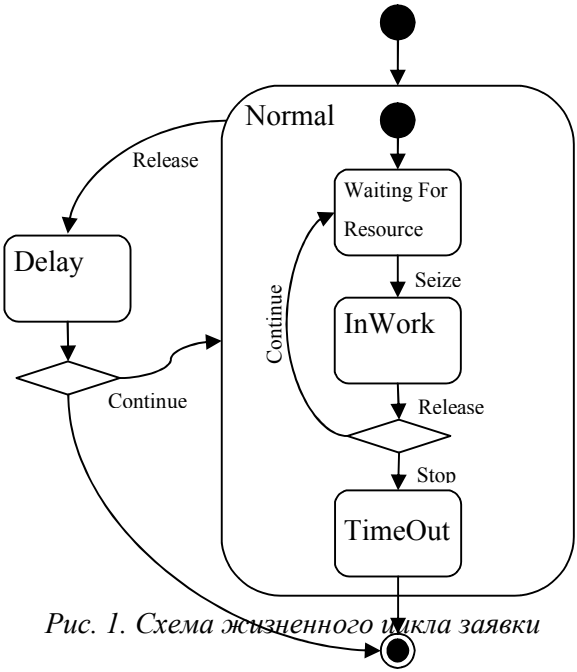

Рис. 1. Схема жизненного цикла заявки

Для каждой заявки можно выделить два основных состояния: в работе (на диаграмме «Normal») и отложено (на диаграмме «Delay»).

Как только появляется новая заявка, она дожидается появления свободного ресурса для ее обработки. При появлении необходимого объема ресурсов, заявка поступает в обработку. После обработки занятые ресурсы освобождаются, и заявка переходит на последний этап – верификация выполнения. В результате верификации определяется степень выполнения заявки и выявляется необходимость каких-либо доработок. В случае, когда необходимы доработки, заявка снова попадает в состояние ожидания свободных ресурсов. Из этого состояния заявка либо возвращается к рабочему циклу, любо завершается вовсе.

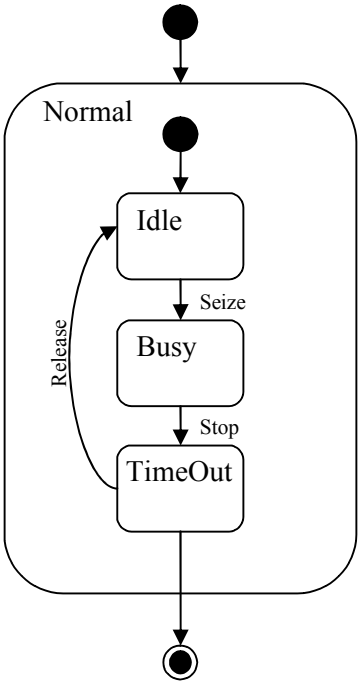

Рис. 2. Схема жизненного цикла обработчика заявок

Поведение агентов, обрабатывающих заявки, можно описать тремя основными состояниями: свободен (Idle), занят (Busy), перерыв (TimeOut). Схема жизненного цикла агента обработчика заявок приведена на рис. 2.

При появлении заявки обработчик переходит в состояние «занят» и выполняет работу. После завершения работ обработчик попадает в состояние «перерыв», так как моментально приступить к обработке новой заявки он не может.

Так как имитация проводится с использованием двух агентов (описывающих заявки и разработчиков), то для дифференциации различной сложности задач и уровня квалификации разработчика введем два коэффициента: коэффициент сложности и коэффициент уровня разработчика. Данные коэффициенты будут влиять не только на время

выполнения работ, но и на стоимость выполнения работ. Введем следующие обозначения:

P_T – время выполнения задачи;

P_D – стоимость выполнения задачи в единицу времени;

K_T – коэффициент сложности задачи;

K_D – стоимость выполнения задачи в единицу времени;

Введем также следующие равенства.

$P_T P_D = 1$ *д.е.* (*денежная единица*)

$K_T P_T = K_T$ *е.в.* (*единиц времени*)

$K_D P_D = K_D$ *д.е./е.в.*

Если работу выполняет разработчик с коэффициентом K_D, то время выполнения задания становится $\frac{P_T}{K_D}$. Из данных равенств получаем следующую зависимость.

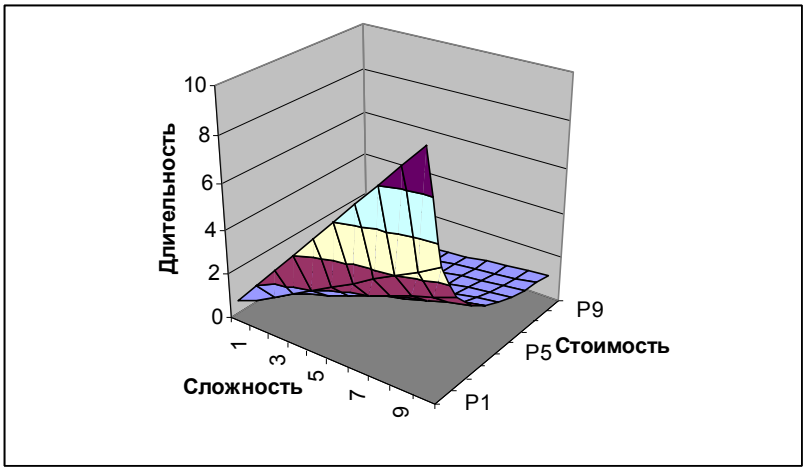

Рис. 3. Зависимость времени выполнения задачи

Таким образом, был завершен последний этап моделирования подсистемы «Разработчик». Результаты имитационных экспериментов проведенных с использованием построенной агентной модели показали неэффективность существовавшего процесса разработки. Применение рекомендаций полученных при анализе результатов

экспериментов, позволили оптимизировать процесс разработки, что благополучно сказалось на качестве программного обеспечения и времени выполнения работ.

Литература

1. АЛИЕВ Т.И. *Основы моделирования дискретных систем* / СПб: СПбГУ ИТМО. – 2009. – 363 с.
2. БАХВАЛОВ Л. *Компьютерное моделирование – длинный путь к сияющим вершинам* // Компьютера. – 1997. – № 40 (217). – С. 26–36.
3. БЕК К. *Экстремальное программирование* // СПб.: Питер. – 2010.
4. БОРЩЕВ А.В. *Практическое агентное моделирование и его место в арсенале аналитика* // Exponenta Pro. – 2004. - № 3-4. – С. 38–47.
5. ГИНЗБУРГ А.И. *Экономический анализ: Предмет и методы. Моделирование ситуаций. Оценка управленческих решений: учебное пособие* / СПб.: Питер. – 2003. – 622 с.
6. КЛЕЙНРОК Л. *Теория массового обслуживания* / Пер. с англ. / Пер. Грушко И.И.; ред. Нейман В.И. / М.: Машиностроение. – 1979. – 432 с.
7. УГОЛЬНИЦКИЙ Г., ТИХОНОВ С. *Имитационное моделирование бизнес-процессов: системы массового обслуживания* / LAP Lambert Academic Publishing. – 2011. – 166 с.
8. ШРАЙБЕР Т.Дж. *Моделирование на GPSS* / Пер. с англ. Гаргера В. Л. , Шмуйлович И.Л.; ред. Файнберг М.А. / М.: Машиностроение. – 1980. – 592 с.
9. ARTHUR, W. B. Designing Economic Agents that Act Like Human Agents: A Behavioral Approach to Bounded Rationality // The American Economic Review. – 1991. – № 81 (2). – P. 353–359.
10. C. MACAL and MICHAEL NORTH. Tutorial on agent based modeling and simulation / Proceedings of the 2005 Winter Si-

mulation Conference, Center for Complex Adaptive Systems Simulations (CAS) // Argonne National Laboratory.
11. ANDREAS WILLING, A Short Introduction to Queuing Theory // Technical University Berlin. – July 21, 1999.

SUBSYSTEM "DEVELOPER" AS PART OF THE RETAIL PAYMENT SYSTEM

Anton Klimenko, Southern Federal University, Rostov-on-Don, Master of Applied Mathematics and Information Technology (antklim@gmail.com).
Guennady Ougolnitsky, Southern Federal University, Rostov-on-Don, Professor and Head, department of Applied Mathematics and Computer Science (ougoln@mail.ru)

Abstract: *In this paper we consider one of the core subsystems of the retail payment system. It's called "Developer". The Queue System for modeling this subsystem was developed and there is the information about it. The task for the assignment problem was set and solved (the modification of the Hungarian algorithm was used). In the conclusion there are the information about Agent Based Model for subsystem "Developer" and the results of the imitation experiments.*

Keywords: Queue System, the Assignment problem, the modification of the Hungarian algorithm, Imitation Modeling, Agent Base Model, Imitation Experiment.