# Аннотация

Данный документ содержит пояснительную записку к модулю «facialExpressionRecognitionLib» («Модуль для мобильных приложений для определения эмоционального отклика по изображению пользователя»), которая служит для создания, просмотра и совместного редактирования интеллект-карт. В разделе «Введение» текущего документа указано наименование программы и документы, на основании которых ведется разработка. Раздел «Назначение и область применения» содержит функциональное и эксплуатационное назначение ПО и краткую характеристику области применения. Раздел «Технические характеристикии» описывает постановку задачи на разработку, описание алгоритмов и схему функционирования программы, методы организации входных и выходных данных, состав технических и программных средств и обоснование выбора алгоритма, метода организации ввода и вывода и состава технических средств. В разделе «Технико-экономические показатели» отражены предполагаемая потребность и экономические преимущества разработки в сравнении с отечественными и зарубежными аналогами. Настоящий документ разработан в соответствии с требованиями:

1. ГОСТ 19.101-77 Виды программ и программных документов [**?**];
2. ГОСТ 19.103-77 Обозначения программ и программных документов [**?**];
3. ГОСТ 19.102-77 Стадии разработки [**?**];
4. ГОСТ 19.104-78 Основные надписи [**?**];
5. ГОСТ 19.105-78 Общие требования к программным документам [**?**];
6. ГОСТ 19.106-78 Требования к программным документам, выполненным печатным способом [**?**];
7. ГОСТ 19.404-79 Пояснительная записка. Требования к содержанию и оформлению [**?**];

# Содержание

# 1 Введение

## 1.1 Наименование программы

Наименование программы: «Приложение для создания и просмотра “интеллект-карты” с возможностью совместной работы по сети в режиме реального времени». Краткое наименование – «Mindmappy».

## 1.2 Документы, на основании которых ведется разработка

Разработка ведется на основании приказа Национального исследовательского университета «Высшая школа экономики» № 2.3-02/1112-04 от 11.12.2019 «Об утверждении тем, руководителей курсовых работ студентов образовательной программы «Программная инженерия» факультета компьютерных наук»

# 2 Назначение и область применения

## 2.1 Назначение программы

### 2.1.1 Функциональное назначение

Основное функциональное назначение программы это создание, редактирование и просмотр диаграмм связей в визуальном редакторе с возможностью совместной работы по сети Internet в режиме реального времени.

### 2.1.2 Эксплуатационное назначение

Функциональным назначением программы является предоставление пользователям интерфейса для редактирования и просмотра диаграмм связей с поддержкой режима коллаборации в реальном времени с использованием двух и более экземпляров программы при наличии сети Internet.

### 2.1.3 Область применения

Программа может быть использована в любой сфере, где необходимо создание интеллект-карт, например описания бизнес-процессов или визуализации работы алгоритма. Режим совместной работы позволит работать над созданием таких карт в команде и упростит процесс коммуникации.

# 3 Технические характеристики

## 3.1 Постановка задачи на разработку программы

Разработать программу, позволяющую создавать, редактировать и просматривать диаграммы связей с помощью графического интерфейса на персональных компьютерах и смартфонах. Разработать библиотеку, позволяющую изменять различные структуры одновременно на нескольких устройствах и синхронизирующую изменения в режиме реального времени. Использовать эту библиотеку для предоставления совместного доступа к редактированию диаграммы связей.

## 3.2 Описание алгоритмов и функционирования программы

### 3.2.1 Описание алгоритмов программы

#### 3.2.1.1 Описание алгоритма синхронизации изменений между клиентами

Для синхронизации изменений между клиентами используется подход YATA [?]. Он реализует CRDT [?] – тип данных, реплицируемый на нескольких клиентах, в котором каждая копия может обновляться независимо от всех остальных, а при синхронизации изменений локально разрешаются все конфликты.

При данном подходе каждый клиент локально хранит у себя редактируемую структуру - документ. Все изменения, совершаемые пользователем, записываются в «вектор состояния» и передаются на удаленные копии документа. Полученные от других клиентов изменения с помощью специального алгоритма интегрируются в локальный документ.

В реализации данного подхода используются следующие сущности:

- Документ (Document) - верхнеуровневая структура, в которой хранятся все остальные подструктуры.
- Структура (Struct) - структура для хранения данных. Возможные типы: Array (массив), Map (ассоциативный массив), Text (текст). В структурах типа Array и Map можно хранить другие структуры, таким образом выстраивая более сложные типы данных.
- Изменение (Item) - объект, описывающий одно атомарное изменение в какой-либо структуре. Такое изменение может добавлять или удалять часть данных в указанной позиции.
- Хранилище (Store) - структура, в которой хранится вектор состояния документа и список еще не обработанных изменений.
- Содержимое (Content) - объект, описывающий содержание изменений и структур. Для реализации программы предусмотрены следующие типы содержимого: строки (ContentString), двоичные данные (ContentBinary) и структуры (Struct).
- Транзакция (Transaction) - объедиенение из нескольких изменений. Для сокращения количества передаваемых данных изменения собираются в транзакцию. После завершения каждой транзакции ее содержимое кодируется и передается другим клиентам, которые могут выполнить эту же транзакцию на своей копии документа, тем самым синхронизируя состояния.

В документе содержатся корневые структуры, к которым можно обращаться по ключу. Создание таких структур не порождает создание объектов типа Item, но в этом нет необходимости, так как на любом клиенте они будут создаваться автоматически при обращении к ним. При создании вложенных структур для каждой новой структуры создается объект изменений.

Изменения могут представлять собой:

1. Вставки. Для Array и Text они характеризуются позицией, для Map – ключом.

2. Удаления. При удалении объекта он помечается меткой deleted, а при необходимости удаленные структуры зачищаются с помощью специального класса GC – сборщика мусора. В данной программе не используется GC, так как он необходим при очень большом количестве удалений и создает дополнительную проблему - если одна из копий не получила все изменения до очистки удаленных структур (например, был отключен интернет), то ее состояние нужно будет сбросить до последнего синхронизированного и могут потеряться локальные изменения.

Внутри себя каждая структура по-своему обрабатывает изменения - например, Map просто вставляет значение по ключу при вставках, а Array и Text работают как обычные списки, но Text оптимизирован для текста и объединяет его последовательные части при «склеивании» событий.

Все изменения хранятся в виде двусвязного списка, отсортированного специальным образом для сохранения единого вида на всех копиях документа. Объекты изменений (Item) имеют следующую структуру:

```
Item {
    id; // структура ID с идентификатором объекта
    left; // ближайшее изменение левее данного
    right; // ближайшее изменение правее данного
    leftOrigin; // ближайшее левое изменение на момент вставки
    rightOrigin; // ближайшее правое изменение на момент вставки
    parent; // родительская структура (Array или Map)
    deleted; // если изменение удаляет данные
    content; // содержимое
    length; // длина содержимого, если она определена, иначе 1
}
```

ID – это уникальный идентификатор измененииия. Он определяется как пара (client, clock), где client - это идентификатор пользователя, который создал изменение, а clock - длина вектора изменений на момент создания нового Item. Такой подход называется «Часы Лэмпорта» [**?**] и используется в распределенных системах.

В случае одного клиента алгоритм внесения изменений выглядит тривиально:

1. находим ближайшее изменение слева и справа от данного

2. если позиция нового изменения находится внутри левого, то разделяем левое изменение на два - левое остается прежним, но с меньшей длинной, а правое создается из остатка левого

3. вставляем новое изменение между левым и правым.

. Но если мы получаем изменения от других клиентов, состояние которых отличалось от нашего, то возникают конфликтные ситуации. Например, если поступило сразу несколько изменений с началом в одной и той же позиции, нужно сохранить их все, при этом таким образом, чтобы их порядок не зависел от порядка их получения, так как на разных клиентах этот порядок может различаться из-за задержки обмена данных по сети.

Рисунок 1 — Мы пытаемся вставить K между двумя T, но в это время появились изменения E и C. Возникла конфликтная ситуация - мы не знаем, в какую из трех позиций поместить новую букву.

Для сохранения единого для всех порядка изменений нужно задать полный порядок на множестве вставок и удалений. Порядок $<_c$ формируется по следущим правилам:

$$o_1 <_{rule1} o_2 \Leftrightarrow o_1 < leftOrigin_2 \vee leftOrigin_2 \leq leftOrigin_1$$

$$o_1 <_{rule2} o_2 \Leftrightarrow \forall o : o_2 <_c o \longrightarrow o_1 \leq o \Leftrightarrow \nexists o : o_2 <_c o < o_1$$

$$o_1 <_{rule3} o_2 \Leftrightarrow leftOrigin_1 \equiv leftOrigin_2 \longrightarrow client_1 < client_2$$

$$o_1 <_c o_2 \Leftrightarrow o_1 <_{rule1} o_2 \wedge o_1 <_{rule2} o_2 \wedge o_1 <_{rule3} o_2$$

Где $o_i$ - какие-то изменения, $leftOrigin_i = o_i.leftOrigin.id.clock$ и $client_i = o_i.id.client$.

То есть, порядок применения изменений зависит от текущего вектора состояний и от id клиента, который внес изменения.

В статье [?] приведено доказательство корректности данного порядка.

Также в статье [?] описан алгоритм, который обеспечивает такой порядок, доказательство его корректности и анализ времени работы. Ниже приведен псевдокод данного алгоритма.

```
// Интеграция изменения Item i в список изменений Item[] i
integrate(i, items) {
    i.left = null;
    for (Item o in items) {
        if ((o.id.clock < i.leftOrigin.id.clock
            || i.leftOrigin.id.clock <= o.leftOrigin.id.clock)
            && (o.leftOrigin.id.clock != i.leftOrigin.id.clock
            || o.id.client < i.id.client))
        {
            // i идет после o
            i.left = o;
        }
        else if (i.leftOrigin.id.clock > o.leftOrigin.id.clock)
        {
            break;
        }
    }
    // задать right и другие свойства
}
```

В данном случае каждая вставка работает за $O(n)$, и чем больше изменений накопилось за время жизни документа, тем медленнее будет работать каждое изменение. Для ускорения работы используются различные оптимизации.

Например, последовательные изменения, имеющие общую границу, склеиваются на локальном документе. Такое склеивание не дает документу быстро разрастаться и сохраняет время работы алгоритма интеграции изменений на уровне $o(n)$, где n - количество всех изменений с момента создания документа, и $O(t)$, где t - размер структуры, к которой мы применяем изменение.

Также при создании нескольких изменений они могут объединяться в транзакцию, таким образом сокращая количество вносимых изменений и размер передаваемых данных.

При передаче сообщений с изменениями они кодируются специальным образом:

```
item = info + leftOrigin.id + rightOrigin + parentId + content
```

"+"обозначает конкатенацию массивов байтов. info - 1 байт, содержащий номер типа и флаги о наличии leftOrigin, rightOrigin и parentId. leftOrigin, rightOrigin и parentId кодируются как пара целых чисел размером 4 байта. content кодируется по-разному в зависимости от его типа. content - закодированное содержание (Content) либо же структура (Struct), которая тоже является контентом, каждый тип кодируется своим образом и содержит 1 байт с информацией о типе.

Далее все изменения для текущей транзакции склеиваются и отправляются на другие копии документа.

На клиенте, получившем изменения, они восстанавливаются и интегрируются в текущий документ. Все изменения выстраиваются в очередь и обрабатываются по принципу FIFO, если получены последовательно, или LIFO, если в обратном порядке – при получении более новых мы должны дождаться предыдущих изменений, если такие были и еще не встроены в текущий документ. Ниже приведена часть кода для «сортировки» полученных изменений и добавления их в конец очереди, если мы не можем их интегрировать.

```
// pendingStack - стек с изменениями, ожидающими обработки
// pendingItems - полученные изменения
// pendingItems.i - текущее обрабатываемое измеенение
// данный код приведен без учета item.id.client

// Пока у нас есть необработанные изменения:
while (pendingStack.Count != 0 || pendingItems.Count != 0)
{
    if (pendingStack.Count == 0) // Если нет изменений для обработки
    {
        var item = pendingItems.First();
        pendingStack.Add(item);
    }
    Item item = pendingStack.Last();
    int localClock = GetState(); // id.clock последнего изменения
    int offset = Min(localClock - item.id.clock, 0);
    // Если мы пропустили изменения до этого:
    if (item.id.clock + offset != localClock)
```

```
    {
        Item nextItem = pendingItems[pendingItems.i++];
        // если нашли изменение с меньшим id.clock, то добавляем его первее
        if (nextItem.id.clock < item.id.clock)
        {
            pendingItems[pendingItems.i] = item; // меняем местами
            pendingStack[pendingStack.Count - 1] = nextItem;
            // удаляем уже обработанные изменения
            // и сортируем остальные по id.clock:
            var newItems = pendingItems.GetRange(
                pendingItems.i,
                pendingItems.Count - pendingRefs.i
            );
            newItems.Sort((r1, r2) => r1.id.clock - r2.id.clock);
            pendingItems = newItems;
            pendingItems.i = 0;
            continue;
        }

        return;
    }
    // missing появляется при получении изменения, left, right или parent
    // которого мы не нашли в нашем документе
    while (item.missing.Count > 0)
    {
        ID missing = item.missing.Count > 0 ? item.missing.Last() : null;
        if (GetState() <= missing.clock)
        {
            // нам нужно получить недостающие изменения:
            pendingStack.Add(pendingItems[pendingItems.i++]);
            break;
        }
        item.missing.RemoveAt(item.missing.Count - 1);
    }
    if (item.missing.Count == 0)
    {
        if (offset < item.length)
        {
            item.Integrate(); // интегрируем изменения
        }
        // мы обработали изменение, можно удалять
        pendingStack.RemoveAt(pendingStack.Count - 1);
    }
}
// У нас еще могли остаться необработанные изменения после этого цикла
// Они будут обработаны при следующих вызовах
```

Данные передаются между клиентами с помощью библиотеки SignalR и простого ретранслирующего сервера.

После внесения локальных изменений или интеграции изменений, полученных от других клиентов, в документе в каждой из измененной структуры вызываются события, которые сигнализируют об изменениях и и указывают на их позицию. Таким образом, мы можем получить полный путь до измененной структуры. Например, если в документе есть структура Array, внутри которой есть структура Map, внутри которой есть структура Text, то можно для каждой из структур, начиная с верхнего уровня, получить позиции изменений и дойти до самой нижней структуры. Для экономии времени можно заранее создать обработчик событий для каждой нужной нам структуры, хоть это и потребует больше памяти. При вызове события изменения в корневом документе текущая транзакция кодируется и передается в двоичном виде, после чего ее сразу можно отправить другим клиентам:

```
document.Update += (changes) => { send(changes); }
```

### 3.2.2 Обоснование выбора алгоритма синхронизации данных

Подход YATA применим не только для совместного редактирования текста, но и для редактирования более сложных типов данных. С помощью вложенных друг в друга массивов и ассоциативных массивов можно легко создать структуру для хранения диаграммы. При этом данный подход является одним из самых эффективных и надежных подходов для совместного редактирования данных. Для библиотеки yjs от автора YATA, реализующей данный подход, проведены измерения скорости работы и использования памяти в сравнении с другими решениями [?].

### 3.2.3 Описание схемы функционирования программы

Диаграма связей внутри программы представлена в виде графа с дополнительной информацией, такой как положение вершин, их форма, размер и подпись. Каждый элемент графа представлен в трех видах:

– геометрический объект;

– графический объект;

– объект для синхронизации между клиентами.

Геометрические объекты - вершины, ребра и сам граф представлены в виде классов из библиотеки Microsoft Automatic Graph Layout [?]. Библиотека позволяет легко добавлять и удалять вершины и ребра, хранить дополнительную информацию и располагать все объекты на плоскости. В библиотеке представлены различные алгоритмы для расчета расположения ребер, в данной программе используется самый быстрый из них – FastIncrementalLayout [?]. Данный алгоритм позволяет создать первоначальное расположение вершин и ребер и изменять положение и форму ребер при перемещении вершин так, чтобы ребра не пересекались с вершинами и были всегда видны.

Графические объекты – фактически обертки над геометрическими объектами для их отображения в окне программы и взаимодействия с ними. Для быстрой отрисовки ребер используется библиотека SkiaSharp [?], которая отличается своей скоростью работы и простотой использования. Для отображения текста и вершин используются обычные графические примитивы платформы UWP [?], которые с помощью фреймворка Uno [?] также отображаются на платформе Android. Вершины можно двигать и

менять размер, добавлять и удалять, менять текст в вершине. Ребра можно добавлять и удалять, менять текст.

Объекты для синхронизации - комбинации из объектов Map, Array, Text, ContentString и ContentBinary из библиотеки для синхронизации. Структура графа представлена следующим образом:

```
nodes: Array
    node: Map
        x: ContentBinary // переводим double в двоичный вид
        y: ContentBinary
        width: ContentBinary
        height: ContentBinary
        text: Text // сразу несколько пользователей могут редактировать текст
edges: Array
    edge: Map
        from: ContentString // неизменяемая строка
        to: ContentString
        text: Text
```

Такие объекты синхронизируются с геометрическими сущностями с помощью событий языка C#. Ниже приведен пример настройки синхронизации для вершин графа:

```
node.PropertyChanged += (sender, e) =>
{
    var text = map.Get("text") as Text;
    switch (e.PropertyName)
    {
        case "Label":
            // TextBox из UWP не отображает атомарные изменения при вводе,
            // поэтому мы считаем diff от текущей и предыдущей версии текста
            text.Diff(node.Label);
            break;
        case "Position":
            map.SetBinary("x", BitConverter.GetBytes(node.Left));
            map.SetBinary("y", BitConverter.GetBytes(node.Top)) ;
            break;
    }
};
map.Update += (sender, changedKeys) =>
{
    foreach (var key in changedKeys)
    {
        switch (key)
        {
            case "x":
            case "y":
                double x = BitConverter.ToDouble(map.GetBinary("x"));
                double y = BitConverter.ToDouble(map.GetBinary("y"));
                node.Left = x;
```

```
                node.Top = y;
                break;
            case "text":
                var text = map.Get("text") as Text;
                node.Label = text.ToString();
    }
};
```

Комбинация из этих трех сущностей позволяет отображать диаграмму на экране и синхронизировать изменения между клиентами в режиме реального времени.

### 3.2.4 Возможные взаимодействия программы с другими программами

Программа для редактирования диаграмм связей работает самостоятельно на двух разных платформах – Android и Windows. Библиотека для совместного редактирования может быть использована в любой другой программе и предоставляет программистам интерфейс для создания безконфликтных реплицируемых типов данных и синхронизации реплик на разных устройствах.

## 3.3 Описание и обоснование выбора метода организации входных и выходных данных

### 3.3.1 Описание метода организации входных и выходных данных

При работе по сети используется библиотека SignalR. При передаче данных структуры документа кодируются способом, описанным в 3.2.1.1. Для синхронизации данных используется сервер-ретранслятор. При подключении нового клиента сервер пересылает полное состояние какого-либо активного клиента. Далее любые изменения в документе передаются всем удаленным копиям в виде транзакций.

### 3.3.2 Обоснование выбора метода организации входных и выходных данных

Библиотека SignalR позволяет обмениваться сообщениями по сети в режиме реального времени. Это популярная библиотека и имеет очень простой интерфейс, что позволяет сократить время на разработку сетевой части программы.

Предложенный способ кодирования используется автором оригинальной статьи [?] и позволяет компактно передавать изменения.

## 3.4 Описание и обоснование выбора состава технических и программных средств

### 3.4.1 Состав технических и программных средств

Для работы программы необходим один из следующих наборов программных средств:

1. операционная система Windows 10 версии 1809 и выше
2. операционная система Android версии 9.0 и выше

Для работы программы на персональном компьютере необходим следующий состав технических средств:

1. Процессор с частотой не менее 1 ГГц
2. Монитор с разрешением 800x600 точек и более;
3. Не менее 1024МБ ОЗУ;
4. Не менее 150МБ свободного места на жёстком диске;
5. Клавиатура;
6. Компьютерная мышь;

Для работы программы на мобильном устройстве необходим следующий состав технических средств:

1. Сенсорный экран с разрешением не менее 320 точек по каждому из измерений;
2. Не менее 512МБ ОЗУ;
3. Не менее 150МБ свободного места на внутреннем накопителе;

Для совместной работы необходимо подключение сети Internet.

### 3.4.2 Обоснование выбора состава технических и программных средств

При реализации программы для ОС Windows использовались возможности платформы UWP, которые поддерживаются в операционной системе Windows 10 версии 1809 и выше. В свою очередь, минимальные системные требования для Windows 10 таковы [**?**]:

1. Процессор с частотой не менее 1 ГГц
2. Не менее 1024МБ ОЗУ;
3. Монитор с разрешением 800x600 точек и более;

Для управления интерфейсом на персональном компьютере предусмотрено и протестировано управление с помощью мыши и клавиатуры. Для управления интерфейсом на смартфоне под управлением Android предусмотрено и протестировано управление с помощью сенсорного ввода.

Минимальное количество памяти ОЗУ, необходимое для работы системы Android 8.0 и выше составляет 512МБ [**?**].

Минимальный размер экрана программы 320x320 точек, поэтому экран любого устройства должен быть с разрешением не менее 320 точек по каждой из сторон.

# 4 Технико-экономические показатели

## 4.1 Предполагаемая потребность

Программа будет использоваться в сфере, где необходима организация знаний или процессов с помощью подхода «Интеллект-карта», например в бизнес-сфере для описания различных процессов или в сфере разработки ПО для описания алгоритма работы программы.

## 4.2 Экономические преимущества разработки по сравнению с отечественными и зарубежными аналогами

Преимущество данной программы в том, что она работает как на персональных компьютерах, так и на смартфонах. Выбранные технические средства, такие как C, Uno и SkiaSharp позволяют легко портировать программу на любые платформы – опытным путем было выяснено, что для работы программы на MacOS, iOS, Linux и в любом браузере с поддрежкой WASM требуется мало изменений в коде. Из аналогов существуют Coggle и Mindomo, работающие только в браузере.

# Список источников

[1] “ГОСТ 19.101-77 Виды программ и программных документов. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001..”

[2] “ГОСТ 19.103-77 Обозначения программ и программных документов. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.”

[3] “ГОСТ 19.102-77 Стадии разработки. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001..”

[4] “ГОСТ 19.104-78 Основные надписи. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001..”

[5] “ГОСТ 19.105-78 Общие требования к программным документам. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001..”

[6] “ГОСТ 19.106-78 Требования к программным документам, выполненным печатным способом. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001.”

[7] “ГОСТ 19.404-79 Пояснительная записка. Требования к содержанию и оформлению. //Единая система программной документации. – М.: ИПК Издательство стандартов, 2001..”

[8] P. Nicolaescu, K. Jahns, M. Derntl, and R. Klamma, “Near real-time peer-to-peer shared editing on extensible data types,” pp. 39–49, 11 2016.

[9] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, “Conflict-free replicated data types,” vol. 6976, pp. 386–400, 07 2011.

[10] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” pp. 558–565, 1978.

[11] “Microsoft automatic graph layout [Электронный ресурс].” // URL: `https://github.com/microsoft/automatic-graph-layout`. (Дата обращения: 15.05.2020, режим доступа: свободный).

[12] “Fastincrementallayout [Электронный ресурс].” // URL: `https://github.com/microsoft/automatic-graph-layout/blob/master/GraphLayout/MSAGL/Layout/Initial/InitialLayout.cs`. (Дата обращения: 15.05.2020, режим доступа: свободный).

[13] “Skiasharp [Электронный ресурс].” // URL: `https://github.com/mono/SkiaSharp`. (Дата обращения: 15.05.2020, режим доступа: свободный).

[14] “Uwp [Электронный ресурс].” // URL: `https://docs.microsoft.com/ru-ru/windows/uwp/get-started/universal-application-platform-guide`. (Дата обращения: 15.05.2020, режим доступа: свободный).

[15] “Uno platform [Электронный ресурс].” // URL: `https://platform.uno/`. (Дата обращения: 15.05.2020, режим доступа: свободный).

[16] “Windows 10 system requirements [Электронный ресурс].” // URL: `https://support.microsoft.com/en-us/help/4028142/windows-10-system-requirements`. (Дата обращения: 15.05.2020, режим доступа: свободный).

[17] “Android 8.0 compatibility definition [Электронный ресурс].” // URL: `https://source.android.com/compatibility/8.0/android-8.0-cdd.pdf`. (Дата обращения: 15.05.2020, режим доступа: свободный).

**ПРИЛОЖЕНИЕ А**

# Используемые понятия и определения

– Диаграмма связей, известная также как интеллект-карта, карта мыслей (англ. Mind map) или ассоциативная карта — метод структуризации концепций с использованием графической записи в виде диаграммы.

– Часы Лэмпорта – фукнция из множества событий распределенных систем в множество неотрицательных целых чисел.

**ПРИЛОЖЕНИЕ В**

# Описание и функциональное назначение классов и структур

Таблица 1 — Классы библиотеки CollabLib

| Класс | Назначение |
|---|---|
| Document | Класс, содержащий структуру для создания документов с возможностью совместного редактирования |
| ID | Класс для обозначения идентификатора изменения или структуры |
| Item | Класс для хранения одного изменения в какой-либо структуре |
| ItemRef | Класс для представления не полностью восстановленных и интегрированных в документ изменений, полученных извне |
| Store | Класс для хранения векторов состояния для пользователей документа. Здесь хранятся все объекты типа Item и очередь изменений. |
| Transaction | Класс для создания транзакций - объединения нескольких изменений для одновременного их выполнения |
| Encoder | Класс для кодирования изменений и состояний в двоичный формат для дальнейшей передачи по сети |
| Decoder | Класс для восстановления структур из двоичных данных |
| AbstractContent | Абстрактный класс для описания данных |
| ContentString | Класс, описывающий данные, содержащие строки |
| ContentBinary | Класс, описывающий двоичные данные |
| AbstractStruct | Абстрактный класс для описания структур данных |
| Array | Класс для создания массивов, интегрируемых в документ для совместного редактирования |
| Map | Класс для создания ассоциативных массивов, интегрируемых в документ для совместного редактирования |
| Text | Класс для создания строк, интегрируемых в документ для совместного редактирования |

Таблица 2 — Классы программы Mindmappy

| | |
|---|---|
| App | Основной класс программы для ее запуска |
| GraphViewer | Основное окно с редактируемой диаграммой |
| Controller | Класс с текущим глобальным состоянием программы |
| EdgesSurface | Класс для отображения и редактирования ребер графа |
| UINode | Класс для отображения и редактирования вершин графа |
| CollabBinding | Класс для привязки библиотеки CollabLib и создания документа с совместным доступом для графа |
| BottomMenu | Класс для обображения нижнего меню программы |

Таблица 3 — Классы программы Server

| | |
|---|---|
| Program | Основной класс программы для ее запуска |
| Startup | Класс с конфигурацией сервера |
| MindmappyHub | Класс с обработчиком событий SignalR и синхронизации изменений между клиентами |