Обнаружение клонов исходного кода: теория и практика

М. Х. Ахин
akhin@kspt.ftk.spbstu.ru

В. М. Ицыксон
vlad@ftk.spbstu.ru

Санкт-Петербургский государственный политехнический университет

При разработке программного обеспечения в условиях жестких временных ограничений программистами часто используется подход «Copy-and-Paste Programming», приводящий к ненужному дублированию исходного кода. Это сильно усложняет такие этапы жизненного цикла программного обеспечения, как тестирование и поддержка. Первым шагом в борьбе с дублированным кодом является его обнаружение.

Данная работа представляет собой краткий обзор методов обнаружения клонов и способов их применения на практике. Проанализированы причины и последствия появления клонов в программном обеспечении. Рассмотрено применение обнаружения клонов для решения таких задач, как нахождение ошибок в дублированном коде, обнаружение клонов в моделях, поиск дублированных фрагментов в HTML. В качестве примеров решения этих задач выбраны подходы, представляющие основные группы методов обнаружения клонов (текстовые, синтаксические, графовые).

Введение

В связи с активным развитием информационных технологий потребности в разработке и поддержке программного обеспечения (ПО) постоянно возрастают с каждым годом. По некоторым

оценкам, общее число строк программного кода удваивается каждый год [9], и это без учета вклада проприетарного ПО.

В этих условиях при разработке ПО активно используется такой подход как *«Copy-and-Paste Programming»* (*CPP*) — копирование и модификация различных фрагментов существующего кода для решения поставленных задач. В подавляющем большинстве случаев CPP является причиной множества проблем на более поздних этапах жизненного цикла ПО.

Одним из способов борьбы с CPP является обнаружение клонов исходного кода ПО (*Software Clone Detection*) — направление программной инженерии, которое активно развивается в настоящее время. Основной целью обнаружения клонов является предоставление разработчику актуальной информации о наличии в программе дублированного кода, который в дальнейшем может быть устранен при помощи рефакторинга.

В рамках обнаружения клонов можно выделить два основных направления развития: теоретическое и прикладное. *Теоретическое* обнаружение клонов ставит своей задачей разработку принципиально новых подходов к обнаружению дублированного кода, не заботясь о применимости этих подходов к реальным программным системам, ограничиваясь проверкой на небольшом множестве тестовых примеров.

Прикладное обнаружение клонов отличается тем, что концентрируется на разработке методов обнаружения клонов, которые могут быть применены на практике для решения определенного круга задач в рамках жизненного цикла разработки ПО. В данной статье мы рассмотрим наиболее интересные задачи из этой области:

- обнаружение ошибок в дублированном коде;
- обнаружение клонов в программных моделях;
- обнаружение дублирования в HTML.

На примере этих задач проанализируем три основные группы подходов к обнаружению клонов исходного кода ПО: текстовые, синтаксические и графовые.

1. Клоны исходного кода программного обеспечения

Многие исследования говорят о том, что довольно значительная часть исходного кода ПО является дублированной. Например, в работе [14] в результате исследований было показано, что от 10 до 15% исходного кода ПО обычно являются клонами. Для эффективного обнаружения клонов исходного кода необходимо хорошо понимать как причины появления клонов в процессе разработки ПО, так и последствия избыточного дублирования кода.

1.1. Причины появления клонов

Выделяют несколько основных причины появления дублированного кода в ПО [4]:

- использование CPP для организации повторного использования кода;
- наличие у программистов «ментальных шаблонов»;
- обеспечение требований к производительности ПО.

В зависимости от конкретного проекта влияние той или иной причины появления программных клонов может варьироваться от незначительной до преобладающей.

1.1.1. Повторное использование исходного кода

В большинстве случаев именно применение CPP становится главной причиной присутствия в ПО избыточного дублирования. Копирование и модификация фрагмента исходного кода является самым быстрым способом повторного использования кода (code reuse). Активное распространение интегрированных сред разработки (IDE, Integrated Development Environment) с развитыми средствами поиска и редактирования исходного кода является одним из стимулирующих факторов к использованию CPP.

Например, при разработке нового модульного теста разработчики часто просто копируют один из имеющихся тестов, после чего адаптируют его под собственные требования. Очевидно, что при наличии большого числа дублированных тестов более правильным

было бы создание абстрактного теста, включающего общую часть тестовой функциональности, с возможностью его параметризации под конкретные тестовые случаи.

1.1.2. Ментальный шаблон

Еще одной причиной появления клонов является возникновение у большинства программистов набора определенных ментальных шаблонов выполнения тех или иных стандартных операций: свертки списка (fold), фильтрации списка (filter) и т. п. Результаты применения таких шаблонов в ходе разработки ПО обычно представляют собой семантические клоны ПО, хотя при этом не производится копирования исходного кода.

Такое дублирование кода является более сложным для обнаружения (по сравнению с CPP), так как при каждом использовании шаблона программист может изменять его в соответствии с требованиями конкретной задачи. Можно сказать, что программист постоянно «изобретает» различные вариации исходного абстрактного ментального шаблона, получая тем самым множество вариантов одного клона исходного кода.

1.1.3. Обеспечение производительности

В некоторых случаях (при наличии жестких ограничений на временные характеристики работы программы) применяются оптимизации, представляющие собой прямое дублирование кода программы. К таким оптимизациям относятся [3]:

- разделение цикла (loop splitting),
- разбиение цикла (loop fission),
- раскрутка цикла (loop unrolling),
- конвейеризация (software pipelining).

В связи с активным развитием технологий оптимизирующих компиляторов, в настоящее время большинство подобных оптимизаций могут быть выполнены автоматически без непосредственного участия человека.

На первый взгляд может показаться, что присутствие дублированного кода не оказывает серьезного влияния на качество ПО,

однако результаты многих исследований в области программной инженерии свидетельствуют об обратном.

1.2. Последствия наличия клонов

Если рассматривать только затраты на первоначальную разработку, то дублирование кода действительно является одним из способов сокращения ресурсов. Однако первоначальная разработка занимает довольно незначительную часть от всего жизненного цикла ПО — большую часть времени (до 80%) занимает поддержка [11]. Из-за этого клоны ПО становятся негативным фактором, способным значительно ухудшить качество программного продукта.

1.2.1. Усложнение модификации кода

Основной причиной, по которой дублирование кода считается плохой практикой программирования, является усложнение модификации ПО при наличии клонов. Это связано с тем, что вместе с явными зависимостями между отдельными частями программы (выраженными, например, в терминах используемого языка программирования) появляются неявные зависимости между дублированными фрагментами кода. При модификации клона ПО, скопированного N раз, требуется найти и изменить все N мест клонирования с учетом возможных отличий разных экземпляров клона.

Поэтому широкое распространение получили различные методологии и принципы, которые помогают избежать возникновения клонов в процессе разработки ПО. Такие правила, как DRY (Don't Repeat Yourself, [12]) или «Правило трех» (Rule of Three, [10]), рекомендуют выполнять рефакторинг дублированного кода сразу после его возникновения. К сожалению, в современных условиях разработки ПО этими правилами зачастую пренебрегают.

1.2.2. Порождение новых ошибок

Кроме усложнения модификации ПО дублирование кода может приводить к появлению новых ошибок. Наиболее частой ситуацией является дублирование кода с небольшой модификацией (например, переименование одной или нескольких переменных), при ко-

тором программист выполнил изменение некорректно (например, забыл переименовать одно из вхождений переменной в дублированный фрагмент). Ряд исследований показали, что подобные ошибки встречаются даже в таких программных проектах, как ядро и драйверы ОС Linux [6, 16].

1.2.3. Увеличение размеров кода

Очевидно, что избыточное дублирование кода влечет за собой увеличение его размеров. Помимо увеличения времени компиляции это приводит к усложнению понимания кода программистом — кратковременная человеческая память оптимально работает с 7 ± 2 объектами [17], а наличие клонов приводит к искусственному снижению этого порога из-за необходимости учитывать дублирование кода.

2. Прикладное обнаружение клонов исходного кода

С точки зрения применения подходов к обнаружению клонов в промышленной разработке ПО наибольший интерес представляет решение с их помощью конкретных прикладных задач. Поэтому в данной работе рассматривается использование обнаружения клонов для:

- обнаружения программных ошибок;
- анализа моделей программ;
- поиска клонов в Web-сайтах[1].

Эти направления программной инженерии активно развиваются в настоящее время, и применение в их рамках обнаружения клонов хорошо иллюстрирует возможности, которые предоставляет современное обнаружение клонов разработчикам ПО. В рамках каждой задачи было выбрано по одной технологии, оказавшей наибольшее

[1]Кроме того, задача поиска клонов актуальна для многих формализованных текстов, в частности, для XML-текстов. Так, в методе DocLine [1], посвященном разработке повторно используемой документации, ставится задача автоматического поиска повторяющихся фрагментов текста при рефакторинге документации, а сама документация задается на XML-подобном языке.

влияние на развитие данного направления, чтобы рассмотреть применение всех трех групп подходов к обнаружению клонов (текстовых, синтаксических, графовых)[2].

2.1. Обнаружение ошибок, связанных с дублированием кода, на примере системы CP-Miner

Наиболее очевидным практическим применением обнаружения клонов является выявление клонов исходного кода, которые были некорректно модифицированы в процессе разработки ПО. Одной из основополагающих работ в этой области является работа [16], в которой рассматривается поиск подобных ошибок в ядре ОС Linux при помощи разработанной авторами системы CP-Miner.

Авторы ставят своей целью решение трех основных задач:

- разработка масштабируемого подхода к обнаружению клонов в ПО;
- обнаружение ошибок, связанных с клонами исходного кода;
- статистическое исследование характеристик дублированного кода.

Для их решения предлагается воспользоваться подходом на основе Frequent Subsequence Mining (FSM) – одного из направлений Data Mining, которое активно развивается в настоящее время [2].

Суть FSM заключается в поиске частых подпоследовательностей символов (items) в базе данных (БД) последовательностей. Подпоследовательность S называется *частой*, если она встречается как минимум в K последовательностях из БД, где параметр K носит название «минимальной поддержки» (minimal support).

Очевидно, что при решении проблемы поиска частых подпоследовательностей «в лоб» будут возникать большие сложности с ресурсоемкостью анализа. Поэтому в CP-Miner используется алгоритм CloSpan [18], обладающий вычислительной сложностью $O(n^2)$ от размера БД последовательностей в случае, когда максимальная длина искомых подпоследовательностей ограничена константой.

[2] К сожалению, ограниченные рамки статьи не позволяют провести более полный и представительный анализ подходов к применению обнаружения клонов ПО на практике.

CloSpan достигает таких показателей за счет двух основных идей. Первая заключается в том, что кандидаты в частые подпоследовательности на каждом шаге алгоритма генерируются при помощи конкатенации частых символов к частым подпоследовательностям, полученным ранее. Если у нас есть множество L_i частых подпоследовательностей длины i и множество L_1 частых символов, то L_{i+1} можно получить как $L_i \times L_1$. Для дальнейшего ускорения работы алгоритма конкатенация элемента из L_i выполняется только с символами, которые входят в БД суффиксов для данного элемента.

Вторая идея, которая применяется для улучшения показателей алгоритма, состоит в оптимизации проверки того, является ли найденная подпоследовательность частой. В качестве оценки числа вхождений подпоследовательности S в БД используется количество различных суффиксов в БД суффиксов для S — в случае, если это число больше заданной минимальной поддержки, то подпоследовательность считается частой.

Для того чтобы FSM можно было использовать для обнаружения клонов, необходимо выполнить отображение исходного кода ПО в формат, над которым можно выполнять поиск частых подпоследовательностей. Каждому оператору программы ставится в соответствие определенное число — в итоге программа представляется в виде множества числовых последовательностей. Для того чтобы можно было находить не только точные, но и изоморфные клоны, отображение операторов программы выполняется без учета именования объектов программы (переменных, функций, пользовательских типов данных и т. д.).

Таким образом, общая схема работы CP-Miner выглядит следующим образом.

1. Преобразование исходного кода в БД последовательностей.
2. Обнаружение кандидатов в клоны исходного кода при помощи FSM.
3. Объединение клонов.
4. Фильтрация ложных обнаружений.

Рассмотрим каждый из этих шагов подробнее.

2.1.1. Преобразование исходного кода

Первым этапом при преобразовании кода является удаление комментариев — в подавляющем большинстве подходов к обнаружению клонов в отношении комментариев используется такая же политика. После этого исходный код программы преобразуется в последовательность чисел и представляется в виде БД последовательностей.

2.1.2. Обнаружение кандидатов в клоны

Для обнаружения клонов применяется алгоритм CloSpan со следующими изменениями, учитывающими характер задачи обнаружения клонов ПО.

1. Обнаружение частых подпоследовательностей с окнами. При копировании и модификации кода довольно частой является ситуация, когда в дублированный фрагмент программы добавляют несколько дополнительных строк. Для учета этого алгоритм CloSpan был расширен параметром *maximal gap* — в случае, если размер окна меньше значения этого параметра, то подпоследовательность считается частой.
2. Отображение частых подпоследовательностей обратно на исходный код. При решении задачи FSM информация о том, в какие последовательности входит та или иная частая подпоследовательность, не требуется; однако обнаружение клонов без обратной связи найденных клонов с исходным кодом является практически бессмысленным. Поэтому при выполнении FSM в CP-Miner с каждой подпоследовательностью дополнительно сохраняется информация о том, где она встречается в исходном коде.

2.1.3. Объединение клонов

Многие из алгоритмов обнаружения клонов работают на определенном уровне гранулярности рассмотрения исходного кода. При этом возникает проблема идентификации связанной группы клонов, состоящей из нескольких последовательных клонов, обнаруженных в процессе анализа. В случае CP-Miner базовой единицей

обнаружения клонов является уровень блока операторов (например, ограниченного фигурными скобками).

Для обнаружения нескольких последовательных клонов используется следующий подход. Для каждого обнаруженного клона проверяется, являются ли клонами соседние фрагменты кода. В случае, если это так, вместо исходного клона в результаты анализа включается составной клон, после чего процедура повторяется. В случае, если клоны-соседи отсутствуют, данный клон помечается как «непоглощаемый» и не участвует в дальнейшем объединении клонов.

2.1.4. Фильтрация ложных обнаружений

Точное обнаружение клонов является практически невозможным из-за огромной вычислительной сложности. Поэтому при анализе ПО возникают ложные обнаружения клонов, которые необходимо отфильтровывать из итоговых результатов анализа. Авторы CP-Miner предлагают несколько эвристик для выполнения фильтрации.

Во-первых, из результатов анализа удаляются все клоны меньше определенного размера. В противном случае (при отсутствии такого ограничения) клонами будут считаться все одинаковые операторы программы — например, `a[i] = C`, — что не является осмысленным с точки зрения обнаружения клонов.

Во-вторых, как было отмечено ранее, в CP-Miner все переменные одного типа считаются сопоставимыми вне зависимости от имени. Однако очевидно, что если между двумя схемами именования переменных для двух экземпляров клона невозможно построить биективное отображение, то возникает конфликт по определенной переменной. Для учета этого вводится такой параметр как *ConflictRatio* — доля несоответствий между именами переменных в клоне. Например, если переменная x была в 80% случаев переименована в y, а в остальных 20% — в z, то $\mathit{ConflictRatio} = 20\%$. В случае, если *ConflictRatio* для клона превышает определенное ограничение, он считается ложным обнаружением.

2.1.5. Поиск ошибок

При поиске ошибок дублирования кода авторы исходят из следующего предположения. Изменение имени переменной между различными экземплярами клона не является ошибкой, так как программист намеренно выполнил это преобразование кода. Ошибкой является ситуация, когда имя переменной было изменено почти во всех экземплярах клона за исключением нескольких, то есть программист забыл выполнить требуемую модификацию.

Для оценки доли подобных ситуаций используется параметр *UnchangedRatio*, рассчитываемый как:

$$UnchangedRatio = \frac{NumUnchanged}{NumTotal},$$

то есть отношение числа неизмененных вхождений переменной в клоны к общему числу вхождений этой переменной. Чем меньше данное отношение, тем более вероятно, что мы имеем дело с ошибкой при модификации кода (за исключением граничного случая, когда $UnchangedRatio = 0$ и все вхождения данной переменной были изменены).

2.1.6. Экспериментальные результаты

Эксперименты показали, что предложенный авторами подход является хорошо масштабируемым — анализ ядра ОС Linux размером более 4 миллионов строк исходного кода занял около 20 минут. В результате CP-Miner обнаружил более 400 потенциальных ошибок модификации, из которых 28 были подтверждены разработчиками ядра ОС Linux как истинные.

В качестве основной причины ложных обнаружений ошибок модификации авторы выделяют изменяемый порядок операторов программы. Например, `a = b; c = d` и `c = d; a = b` являются семантически эквивалентными блоками программы, однако CP-Miner может обнаружить в данном случае ошибку модификации.

2.1.7. Резюме

Результаты применения CP-Miner на практике показывают, что создание средств для поддержки дублированного кода, способных

обнаруживать ошибки модификации, действительно является весьма актуальной задачей. Поиск и исправление подобных ошибок представляет серьезную проблему без использования методов обнаружения клонов, так как большинство классических подходов к обнаружению программных ошибок рассчитаны на другие классы дефектов.

2.2. Обнаружение клонов в моделях на примере анализа моделей Matlab

Активное развитие в последние 5–10 лет получило такое направление программной инженерии, как *Model Driven Development* (*MDD*). Например, в области разработки бортовых автомобильных систем до 80% исходного кода может генерироваться из специализированных моделей [13]. Для таких моделей можно выделить ряд проблем, характерных и для обычных языков программирования, например, присутствие клонов в описаниях различных моделей. Поэтому в последние годы начало развиваться направление обнаружения клонов в моделях. Одной из первых работ в этой области является работа [8], в которой рассматривается задача поиска дублированных фрагментов моделей Matlab/Simulink/TargetLink.

Анализу подвергается графовая модель, получаемая из исходной модели Matlab. Для этого выполняется следующая предобработка модели:

- подстановка тел всех модулей в места использования;
- удаление несвязанных компонентов;
- нормализация — разметка всех вершин и дуг графа в соответствии с требуемой для обнаружения клонов информацией.

В результате модель Matlab представляется в виде ориентированного графа $G = (V, E, L)$, где V — множество вершин графа, E — множество дуг графа, $L : V \cup E \to N$ — отношение, ставящее в соответствие вершинам и дугам графа метки из заданного множества меток N.

После такого преобразования задача обнаружения клонов в модели сводится к задаче поиска изоморфных подграфов G_1 и G_2 в графе G со следующими дополнительными ограничениями:

- G_1 и G_2 являются максимальными изоморфными подграфами, то есть не существует другой пары изоморфных графов, которые полностью включают в себя G_1 и G_2;
- G_1 и G_2 не имеют общих вершин, то есть $V_1 \cap V_2 = \emptyset$;
- G_1 и G_2 являтся связанными, то есть для любой пары вершин существует путь между ними.

Данные ограничения не влияют на тот факт, что задача поиска изоморфных подграфов является обобщением задачи о клике (Maximum Clique Problem), которая относится к классу NP-полных задач [15]. Поэтому для решения подобных задач на практике применяются различные эвристические подходы, позволяющие получить решение задачи за приемлимое время.

В [8] авторы предлагают сократить сложность задачи поиска изоморфных графов за счет рассмотрения на каждом шаге только одного возможного варианта биективного отображения между соседними вершинами. При классическом подходе к поиску изоморфных подграфов при рассмотрении пары вершин v_1 и v_2 необходимо проанализировать все возможные варианты отображений между соседними для v_1 и v_2 вершинами, что приводит к экспоненциальному росту сложности решения задачи. Для того, чтобы решить, какой вариант отображения будет выбран на каждом шаге, вводится функция близости вершин.

Функция близости $\sigma : V \times V \to [0, 1]$ рассчитывается на основе похожести не только самих вершин v_1 и v_2, но и их соседей. Для этого используется вспомогательная функция $s_i : V \times V \to [0, 1]$, определенная следующим образом:

$$s_0(u, v) = \begin{cases} 1 & L(u) = L(v) \\ 0 & otherwise \end{cases},$$

$$s_{i+1}(u, v) = \begin{cases} \frac{M_i(u,v)}{max(|N(u)|,|N(v)|)} & L(u) = L(v) \\ 0 & otherwise \end{cases},$$

где $N(v)$ – множество соседних к v вершин, $M_i(u, v)$ – максимальный возможный вес отображения между $N(u)$ и $N(v)$, рассчитываемый на основе s_i.

В таком случае σ можно задать как:

$$\sigma(u, v) = \sum_{i=0}^{\infty} \frac{1}{2^i} s_i(u, v).$$

На практике можно ограничиться вычислением нескольких первых компонентов суммы с использованием динамического программирования для снижения вычислительной сложности алгоритма.

2.2.1. Группировка результатов

После обнаружения клонов мы получаем множество пар клонов. В случае, если фрагмент модели был продублирован N раз, будет получено $\frac{N(N-1)}{2}$ клоновых пар. Поэтому необходимо выполнить их группировку для более удобного отображения результатов анализа.

При группировке вместо очевидного решения объединять вместе идентичные клоны авторы предлагают объединять клоны только на основе равенства множества вершин, входящих в них, без учета соединяющих их дуг. Для моделей Matlab состав блоков является более важным, чем схема их соединения друг с другом, что и является причиной такого решения.

2.2.2. Экспериментальные результаты

Проверка предложенного подхода проводилась на реальной модели системы управления силовым блоком, используемой в компании MAN AG и состоящей из более чем 20 тысяч блоков TargetLink. Было найдено более 300 клонов, из которых около 50% представляют собой дублирование вспомогательных элементов модели, таких как терминаторы и мультиплексоры.

Для решения проблемы дублирования кода уже после экспериментов было предложено ввести дополнительные эвристики для фильтрации обнаруженных клонов:

- все клоны небольшого размера (меньше 5 блоков) удаляются из результатов анализа;
- каждому типу блока модели ставится в соответствие определенный вес — для вспомогательных элементов этот вес равен

нулю. После этого все клоны с общим весом меньше заданного ограничения не попадают в рассмотрение.

После экспертного анализа оставшихся после фильтрации клонов специалистами MAN AG многие из них были признаны истинными и включены в состав библиотеки компонентов для дальнейшего использования. Кроме того, анализ библиотеки, используемой при разработке модели силового блока, показал, что в ее составе также присутствует определенное число клонов, то есть при разработке библиотеки одна и та же функциональность была независимо реализована несколько раз.

2.2.3. Резюме

Обнаружение клонов в моделях является новым направлением, которое еще только начинает свое развитие. Несмотря на это, результаты исследований в этой области позволяют говорить о том, что обнаружение клонов в моделях может использоваться на практике для повышения качества разрабатываемого ПО. С учетом современной тенденции к автоматизации разработки ПО при помощи генерации кода по моделям можно сказать, что данное направление обнаружения клонов является одним из самых перспективных для дальнейших исследований.

2.3. Обнаружение клонов в HTML при помощи TXL

При разработке Web-сайтов в условиях крайне ограниченного времени разработчикам приходится прибегать к дублированию HTML-кода из-за отсутствия в HTML механизмов повторного использования кода. Более того, часто Web-разработчики просто не знакомы с хорошими практиками программирования и рассматривают дублирование кода как один из основных инструментов при разработке [5]. В связи с этим обнаружение клонов в HTML представляет несомненный практический интерес.

В принципе, HTML можно рассматривать как обычный текст, поэтому для поиска клонов в HTML применимы многие классические методы обнаружения клонов, основанные на простом текстовом анализе. Одной из наиболее интересных работ в этой области

является работа [7], в которой обнаружение HTML-клонов рассматривается как основная решаемая задача.

Предлагаемый подход основан на TXL, которая представляет собой систему трансформации исходного кода на основе специальных правил преобразования. С ее помощью авторы предлагают форматировать исходный код таким образом, чтобы его можно было эффективно сравнивать при помощи стандартной утилиты для сравнения текстовых файлов *diff*.

Для этого выполняются следующие трансформации кода:

- нормализация форматирования — удаление комментариев и лишних символов пробельной группы, приведение к единому стилю форматирования HTML-тегов;
- локализация изменений на отдельных строках — все семантически значимые элементы HTML (теги, атрибуты тегов, параметры HTML) по возможности располагаются на отдельных строках.

Таким образом, исходный код представляется в виде, удобном для построчного сравнения при помощи *diff*.

Нормализованный исходный код после этого поступает на вход утилиты *diff*. Для каждого файла выполняется его сравнение с другими файлами и определяется доля уникальных строк одного файла относительно другого и наоборот. В случае, если оба значения ниже заданного ограничения, эта пара файлов образует клон.

Таким образом, вычислительная сложность алгоритма равна $O(n^2)$, где n — число файлов в анализируемой Web-системе. Для оптимизации было предложено использовать ряд допущений, чтобы сократить число требуемых сравнений:

- сравнение файлов выполняется только в случае, если размер одного файла не более чем в 2 раза превосходит размер другого;
- после обнаружения файлов-клонов в дальнейшем сравнении участвует только один из представителей данного клонового класса.

Можно отметить, что такой подход может быть легко адаптирован для поиска клонов не только в HTML, но и в исходном коде любого языка программирования, так как он не использует никаких знаний о синтаксисе или семантике анализируемого текста.

2.3.1. Экспериментальные результаты

Тестирование подхода проводилось на двух Web-сайтах среднего размера в 10 тысяч строк HTML-кода. Время анализа составило 3–5 минут; в результате было обнаружено несколько десятков клоновых классов в каждом случае.

Обнаруженные HTML-клоны в большинстве своем представляют собой следующие группы:

- клоны Web-интерфейса;
- клоны форматирования отдельных элементов сайта;
- клоны отдельных HTML-страниц.

Для всех этих групп может быть выполнен рефакторинг, который устранит избыточное дублирование HTML-кода.

2.3.2. Резюме

Различные области IT так или иначе имеют дело с текстовой информацией — Web-разработка, биоинформатика, компьютерная лингвистика и т. п. При возникновении необходимости обнаружения общих элементов (клонов) без учета особенностей конкретной предметной области могут использоваться практически любые текстовые методы обнаружения клонов исходного кода ПО.

2.4. Итоги

По результатам рассмотрения данных трех примеров применения обнаружения клонов на практике можно сделать следующие выводы.

1. Центральным параметром обнаружения клонов является используемое модельное представление анализируемой предметной области, от которого зависят используемые методы обнаружения дублирования.
2. При применении обнаружения клонов на практике целесообразным является использование различных эвристик, зависящих от особенностей предметной области.
3. Для удобной и эффективной работы с результатами анализа необходимо использовать соответствующие способы их визуализации — например, объединение обнаруженных клонов в группы или кластеры.

Заключение

Данная работа является кратким обзором методов обнаружения клонов исходного кода ПО на практике. Представлены основные причины появления клонов и проблемы, связанные с дублированным кодом. Рассмотрено прикладное применение обнаружения клонов для решения трех задач: поиск ошибок модификации дублированного кода на примере системы CP-Miner, обнаружение клонов в моделях бортовых автомобильных систем на языке Matlab, поиск дублирования в HTML-страницах при помощи TXL. На данных примерах рассмотрены все основные классы подходов к обнаружению клонов: текстовые, синтаксические и графовые, а также их характерные особенности.

Список литературы

[1] *Кознов Д. В., Романовский К. Ю.* DocLine: метод разработки документации семейства программных продуктов // Программирование. Вып. 4. 2008. С. 1–13.

[2] *Agrawal R., Srikant R.* Mining Sequential Patterns // Proceedings of the Eleventh International Conference on Data Engineering. IEEE Computer Society. 1995. P. 3–14.

[3] *Bacon D. F., Graham S. L., Sharp O. J.* Compiler Transformations for High-Performance Computing // ACM Computing Surveys. Vol. 26, N 4. ACM. 1994. P. 345–420.

[4] *Baxter I. D., Yahin A., Moura L. e. a.* Clone Detection Using Abstract Syntax Trees // Proceedings of the International Conference on Software Maintenance. IEEE Computer Society. 1998. P. 368.

[5] *Brereton P., Budgen D., Hamilton G.* Hypertext: The Next Maintenance Mountain // Computer. Vol. 31, N 12. IEEE Computer Society. 1998. P. 49–55.

[6] *Chou A., Yang J., Chelf B. e. a.* An Empirical Study of Operating Systems Errors // Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles. ACM. 2001. P. 73–88.

[7] *Cordy J. R., Dean T. R., Synytskyy N.* Practical Language-Independent Detection of Near-Miss Clones // Proceedings of the 2004

Conference of the Centre for Advanced Studies on Collaborative Research. IBM Press. 2004. P. 1–12.

[8] *Deissenboeck F., Hummel B., Juergens E. e. a.* Clone Detection in Automotive Model-Based Development // Proceedings of the 30th International Conference on Software Engineering. ACM. 2008. P. 603–612.

[9] *Deshpande A., Riehle D.* The Total Growth of Open Source // Proceedings of the Fourth Conference on Open Source Systems. IFIP. Vol. 275. Springer Verlag. 2008. P. 197–209.

[10] *Fowler M., Beck K., Brant J. e. a.* Refactoring: Improving the Design of Existing Code. Addison-Wesley Professional. 1999. 464 p.

[11] *Glass R. L.* Frequently Forgotten Fundamental Facts about Software Engineering // IEEE Software. Vol. 18, N 3. IEEE Computer Society. 2001. P. 110–112.

[12] *Hunt A., Thomas D.* The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley Professional. 1999. 352 p.

[13] *Jungmann M., Otterbach R., Beine M.* Development of Safety-Critical Software Using Automatic Code Generation // Proceedings of SAE World Congress. 2004.

[14] *Kapser C., Godfrey M. W.* Toward a Taxonomy of Clones in Source Code: A Case Study // Evolution of Large-scale Industrial Software Applications (ELISA). 2003. P. 67–78.

[15] *Karp R. M.* Reducibility Among Combinatorial Problems // Complexity of Computer Computations. Plenum Press. 1972. P. 85–103.

[16] *Li Z., Lu S., Myagmar S., Zhou Y.* CP-Miner: a Tool for Finding Copy-Paste and Related Bugs in Operating System Code // Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation. USENIX Association. 2004. P. 20.

[17] *Miller G. A.* The Magical Number Seven, Plus or Minus Two // The Psycological Review. Vol. 63. 1956. P. 81–97.

[18] *Yan X., Han J., Afshar R.* CloSpan: Mining Closed Sequential Patterns in Large Datasets // Proceedings of SIAM International Conference on Data Mining. 2003. P. 166–177.

[19] *Yarmish G., Kopec D.* Revisiting Novice Programmer Errors // ACM SIGCSE Bulletin. Vol. 39, N 2. ACM. 2007. P. 131–137.