Программная инженерия

SWEBOK KA-13:

Computing Foundations

Вопросы

- Абстракции
- Программирование, алгоритмы, представление данных
- Отладка, компиляция, разрешение проблем
- Компьютеры, ОС
- Сети
- Параллельные вычисления
- ~~Все люди сво~~ Человеческий фактор
- Безопасность

Зачем

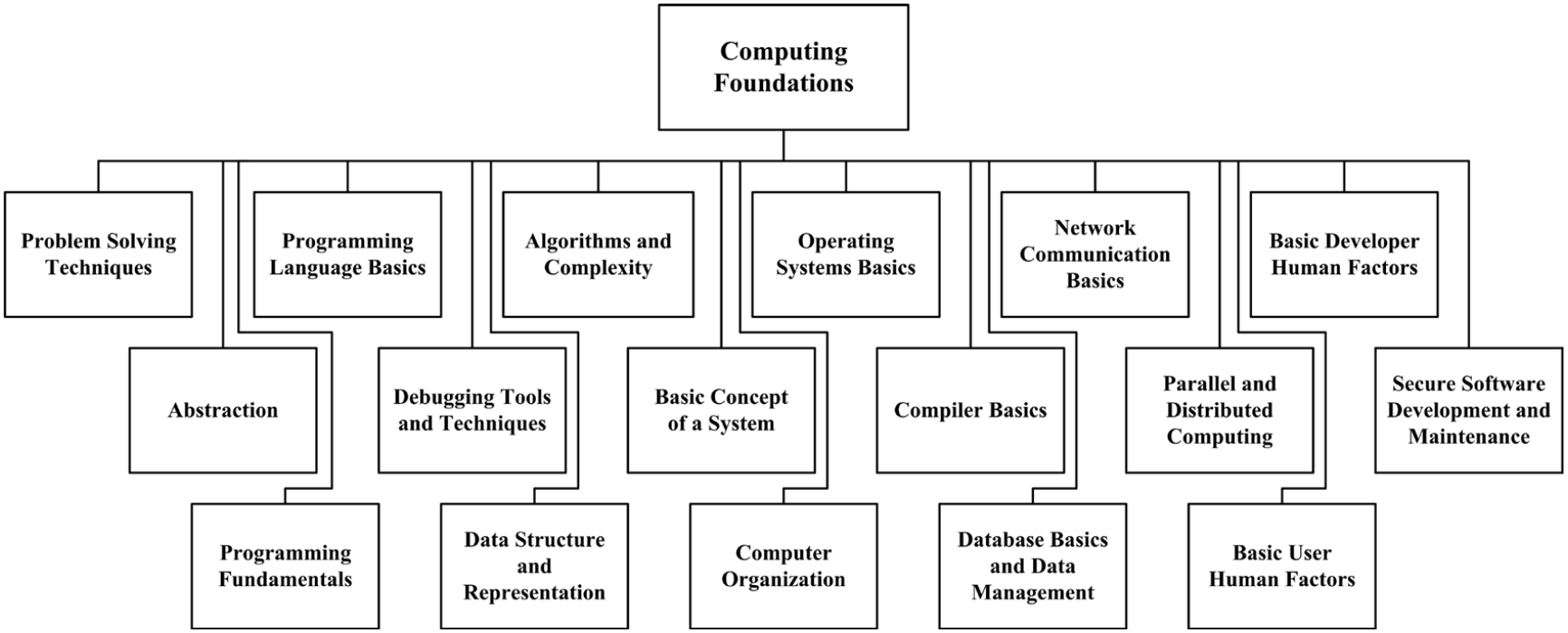
Состав раздела

SWEBOK: COMPUTING FOUNDATIONS (КА-13)

Причина введения КА

- Данный КА описывает среду, в которой ПО развивается (evolves) и выполняется
- Большая часть тем, затронутых в КА, также обсуждается в курсах специальных дисциплин. Однако, у таких дисциплин есть недостатки:
  - Часть тем из них не является связанной и/или важной для ПИ. Например, компьютерная графика
  - Часть тем не выделена как отдельные дисциплины и/или «размазаны» по нескольким дисциплинам

Структура КА



Практические замечания

- Всего в КА выделено 17 тем
- Для каждой будет приведено:
  - Содержание согласно КА
  - Ссылка на конкретные дисциплины и/или источники дополнительной информации

Общий подход

Формулирование

Анализ

Стратегии поиска решения

РАЗРЕШЕНИЕ ПРОБЛЕМ

Общий подход

- Разрешение проблем называется обдумывание и операции (activities), приводящие к получению ответа или выработке решения проблемы
  - Например, ПИ концентрируется на решение проблем с помощью компьютеров и ПО
- В общем случае, выделяется три этапа разрешения проблемы:
  - Формулировка [проблемы]
  - Анализ [проблемы]
  - Разработка стратегии поиска решения

Формулирование

- Сформулированная проблема должна явно указывать собственно проблему и желаемый результат ее разрешения
- Нет универсального способа. Среди наиболее общих подходов выделяют:
  - Определение источника и причины [проблемы]
  - Переформулирование проблемы
  - Анализ текущего и желаемого состояния
  - «Свежий взгляд» (fresh eye approach)

Анализ – 1/2

- Анализ проблемы помогает структурировать поиск решения
- Использует следующие виды анализа:
  - Ситуационный
    - Сначала исследуются наиболее критические или срочные аспекты ситуации
  - Проблемный (problem analysis)
    - Определяются причины возникновения проблемы
  - Анализ решений (decision analysis)
    - Определяются действия, необходимые для решения проблемы или исключения причин ее возникновения

Анализ – 2/2

- Виды анализа (продолжение):
  - Анализ потенциальных проблем (potential problem analysis)
    - Определяются действия, необходимые для предотвращения повторного возникновения старой проблемы или появления новых проблем

Разработка стратегии поиска решения

- «Лучшее» решение может быть определено различным образом:
  - Быстрее
  - Дешевле
  - Более легкое в использовании
  - Имеющее новые функции, etc.
- Необходимо:
  - Исключить пути ведущие к менее «хорошим» решениям
  - Определять задачи таким образом, чтобы направлять поиск в верном направлении
  - Установить характеристики конечного решения, показывающие его «хорошесть»

Использование компьютера

- Основные вопросы:
  - Как объяснить компьютеру что надо сделать?
  - Как представить описание проблемы в машинно-читаемом виде?
  - Как описать процесс решения проблемы на ЯП?

Уровни

Инкапсуляция

Иерархия

Альтернативы

АБСТРАКЦИИ

Абстракция

- Абстракция незаменима в решении проблем. Она применима как к процессу решения, так и к получаемому результату путем уменьшения количества одновременно «видимой» информации о концепции, проблеме, наблюдаемом явлении → «big picture»
- Важным инженерным навыком является умение выбрать верный уровень абстракции
- Абстракция позволяет распознать отношения между различными аспектами проблемы и найти более эффективное ее решение

Уровни абстракции – 1/2

- Вводя абстракцию мы сосредотачиваем внимание на одном уровне за раз, принимая что мы можем эффективно соединить этот уровень с уровнями выше и ниже
  - Однако, это не значит, что при выделении уровня мы ничего не «знаем» о соседних уровнях

Уровни абстракции – 2/2

- Обычно уровень абстракции соответствует не отдельному компоненту системы, а некоему «уровню представления» нижележащих элементов. Примеры:
  - Подсистемы
  - Сетевые уровни OSI ISO
- Взаимодействие между уровнями задает интерфейс (например API). Преимущество стандартизованных интерфейсов уровней заключается в облегчении переносимости и упрощении интеграции в дальнейшем

Инкапсуляция

- Инкапсуляция – это механизм реализации абстракции. Когда мы выбираем уровень абстракции уровни выше и ниже инкапсулируются
  - Инкапсулируемой информацией может быть концепция, проблема, наблюдаемое явление. Допустимые операции с ними – интерфейс
  - Обычно прячут часть информации о других уровнях (упрощая ее). Например, подробности устройства или способы выполнения запрашиваемых действий

Иерархия – 1/2

- При работе с информацией мы используем в разное время разные уровни абстракции. Обычно, эти разные уровни образуют иерархию
  - Выделение иерархии процесс индивидуальный и результат зависит от исполнителя
- Иерархия является естественным результатом декомпозиции
  - Анализ задач приводит к созданию иерархии задач и подзадач (WBS – work breakdown structure)

Иерархия – 2/2

- Варианты иерархий:
  - Последовательная
    - Ровно один уровень выше и ниже (исключая крайние)
  - (Чаще) Древовидная
  - (Редко) Многие-ко-многим
  - (Никогда!) Циклическая

Альтернативы

- Бывает полезным (практически всегда – прим. преподавателя) иметь одновременно несколько различных абстракции одного явления для того, чтобы рассмотреть его с разных сторон (perspectives). Например:
  - Диаграмма классов
  - Диаграмма состояний
  - Диаграмма последовательности
- Альтернативы не образуют иерархию, но дополняют друг друга для более полного описания явления

PROGRAMMING FUNDAMENTALS

Основы программирования

- Процесс разработки программ описан (внезапно!) в ПИ
- Парадигмы программирования:
  - Неструктурированное
  - Структурированное (процедурное, императивное)
  - Функциональное
  - ООП
  - Аспектно-ориентированное (aspect oriented programming)

Aspect-oriented programming

- Строится поверх ООП
- Цель АОП – изолировать вторичные и/или обеспечивающие функции от основной бизнес-логики приложения
- Основная цель АОП – избежать запутывания, вносимого ООП в виде детального описания очень сложных взаимодействий объектов

ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Классификация

- По «уровню» (абстрагированию от компьютерного представления)
  - Низкоуровневые
    - Пример: ассемблер
  - Высокоуровневые
    - Пример: C++, Java
- По отношению к способу инициации вычислений
  - Декларативные
    - Пример: Prolog, Lisp
  - Императивные

СРЕДСТВА И ТЕХНИКИ ОТЛАДКИ [КОДА]

Типы ошибок

- Синтаксические
  - Самые «простые» из видов ошибок – обнаруживаются компилятором. Нарушения синтаксиса могут «перерождаться» в логические ошибки
- Логические
  - Семантические ошибки, приводящие к неверным вычислениям или неправильному поведению ПО
  - Не отлавливаются компилятором. Часть ошибок может быть выявлена статическими анализаторами кода
- Ошибки данных
  - Ввод неверных данных (неправильного формата или «не тех» данных)

Отладка

- Статическая
  - Ревизия кода
- Динамическая
  - Трассировка выполнения
- После падения
  - Анализ core dump «упавшего» процесса
- Трассировка – выполнение кода с просмотром содержимого памяти (переменных, регистров, etc)
  - Пошаговое выполнение
  - Использование точек останова (breakpoints)
  - Отслеживание изменений переменных (watch points)

Средства отладки

- Отладчики
  - Используются для динамической отладки
  - Встроены во все современные IDE
  - Функции: отслеживание выполнения, запуск / перезапуск процесса, остановка выполнения, работа с точками останова и watch points, прямое редактирование переменных (памяти), иногда, сдвиг времени назад
- Анализаторы кода
  - Используются для статической отладки
- Просмотрщики core dump

СТРУКТУРЫ ДАННЫХ И ПРЕДСТАВЛЕНИЕ

Структуры данных – 1/2

- Структуры данных это абстракции, предназначенные для сбора собственно данных и связанных с ними операций
- Специфические структуры данных часто создаются специально для:
  - Увеличения эффективности программ и алгоритмов. Например:
    - Стеки
    - Очереди
    - Кучи
  - Обозначения концептуальной сущности, объединяющей разные данные. Например:
    - Имя и адрес человека

Структуры данных – 2/2

- Выбор правильной структуры данных может означать выбор между выполнением кода ее использующего за секунды или часы и даже дни
- Структуры данных можно разделить на:
  - Линейные / нелинейные
  - Гомогенные / гетерогенные
  - Статические / динамические
  - Временные / постоянные
  - Внешние / внутренние
  - Примитивные / агрегированные
  - Рекурсивные / нерекурсивные
  - Пассивные / активные
  - С состоянием / без состояния

Типы структур: линейные/нелинейные

- Линейные структуры данных упорядочивают элементы в одном измерении:
  - У каждого элемента есть не больше одного предшественника и следующего за ним элемента
  - Если предшественника нет, то элемент первый
  - Если следующего элемента нет, то элемент последний
  - Примеры: списки, стеки, очереди
- Нелинейные структуры организуют элементы в двух и более измерениях
  - У каждого элемента может быть любое число предшественников и следующих элементов
  - Примеры: кучи, хэш-таблицы, деревья (бинарные, сбалансированные, B и B+ деревья, и др.)

Типы структур: compound structure

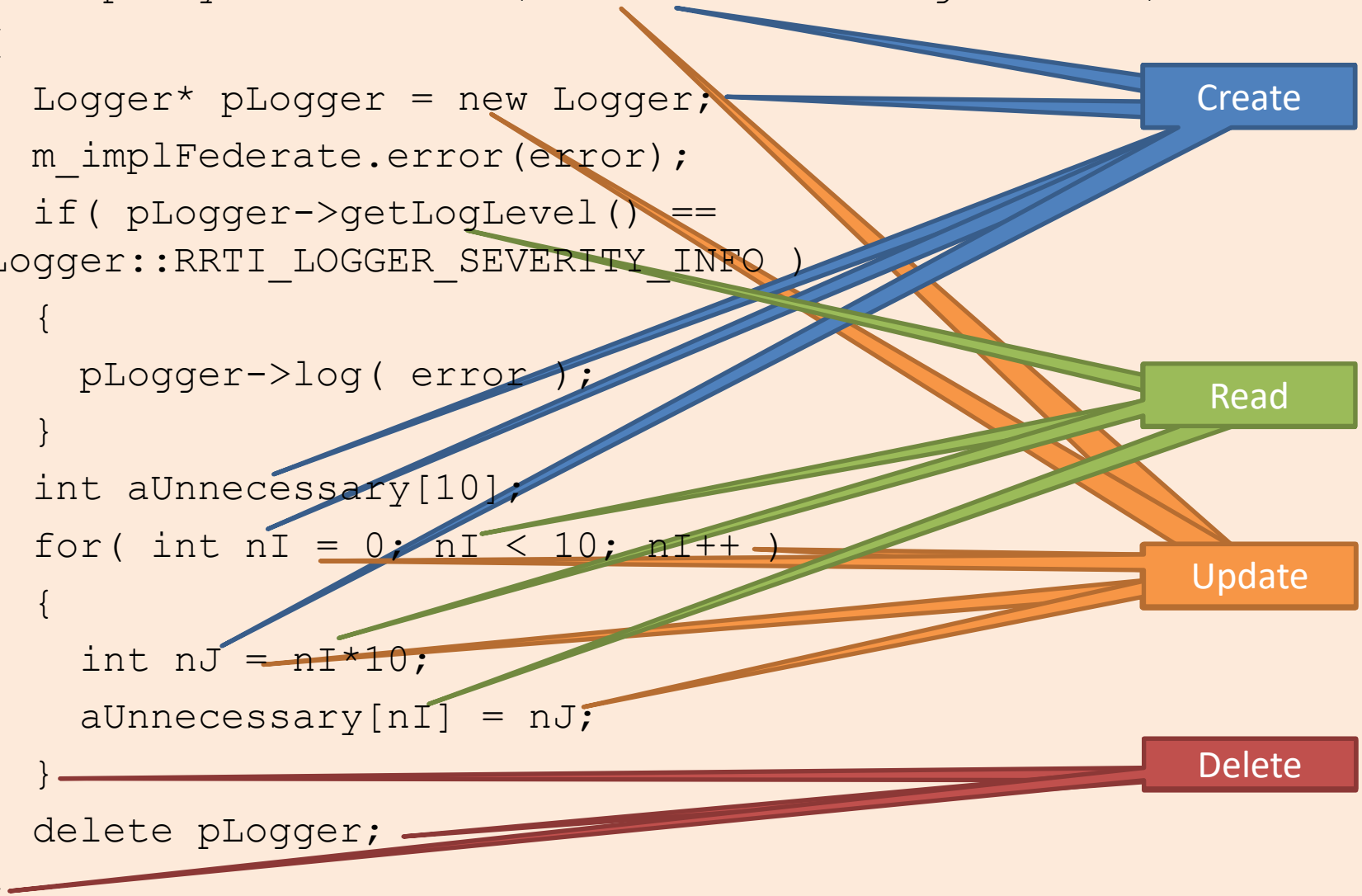
- Составные структуры (compound structure) строятся из других (более примитивных) структур данных
  - В самом «низу» иерархии находятся простые или «примитивные» типы данных (логические, целые и др.)
  - Составная структура может содержать другие составные структуры, в том числе рекурсивно
  - Примеры составных структур: наборы, графы и разделы. Например, раздел можно рассматривать как набор наборов

Базовые операции над структурами

- Все структуры поддерживают базовый набор операций (create, read, update, and delete – CRUD):
  - Create – добавляет элемент в структуру
  - Read – возвращает элемент из структуры
  - Update – изменяет существующий элемент в структуре
  - Delete – удаляет элемент из структуры

Врезка: найди операцию – 1/4

```
void proxyModel::error( const std::string& error )
{
  Logger* pLogger = new Logger;
  m_implFederate.error(error);
  if( pLogger->getLogLevel() ==
Logger::RRTI_LOGGER_SEVERITY_INFO )
  {
    pLogger->log( error );
  }
  int aUnnecessary[10];
  for( int nI = 0; nI < 10; nI++ )
  {
    int nJ = nI*10;
    aUnnecessary[nI] = nJ;
  }
  delete pLogger;
}
```



Врезка: найди операцию – 2/4

- Create/Delete:
  - Просто:
    - Явное создание и уничтожение переменной pLogger
  - Чуть сложнее:
    - Определение массива aUnnecessary → удаляется автоматически при завершении метода
    - Определение в заголовке цикла переменной nI → удаляется автоматически после завершения цикла
    - Определение в теле цикла переменной nJ → создается и удаляется при каждом повторе цикла
  - Неочевидно:
    - Строка error передается как константный параметр, поэтому создается копирующим конструктором при вызове метода и удаляется при выходе из него

Врезка: найди операцию – 3/4

- Read:
  - Просто:
    - Значение переменной nI сравнивается с пределом после каждого повтора цикла
    - Значение переменной nI используется для установки значения переменной nJ
    - Индекс массива устанавливается равным значению переменной nI
  - Чуть сложнее:
    - Значение уровня вывода в лог (переменная-член класса экземпляра pLogger класса Logger) доступно как результат вызова метода getLogLevel() (классический getter)

Врезка: найди операцию – 4/4

- Update:
  - Просто:
    - Значение переменной nI устанавливается равным 0 после создания
    - Значение переменной nI увеличивается на единицу после каждого повтора цикла
    - Значение переменной nJ устанавливается после ее создания
    - В элемент массива с номером nI записывается значение переменной nJ
  - Чуть сложнее:
    - Переменная pLogger после создания устанавливается равной указателю на экземпляра класса Logger в результате вызова конструктора класса в явном виде
  - Сложно:
    - Значение параметра error устанавливается в результате неявного вызова копирующего конструктора

Операции над структурами данных

- Кроме CRUD некоторые структуры поддерживают дополнительные операции:
  - Найти конкретный элемент структуры
  - Упорядочить элементы структукры согласно некоторым правилам
  - Обойти все элементы в определенном порядке
  - Реорганизовать или «перебалансировать» структуру
    - Rebalance согласно кембриджскому словарю: to change the amount or level of one or more things in order to improve a particular situation

Эффективность операций

- Различные структуры поддерживают разные операции с различной эффективностью. Различие в эффективности может быть значительным. Например:
  - Удаление последнего добавленного элемента эффективно в стеке
  - Удаление элемента крайне неэффективно в массиве
  - Поиск поддерживается контейнером map
  - Добавление в контейнер map происходит быстро, а удаление – медленно

АЛГОРИТМЫ И [ИХ] СЛОЖНОСТЬ

Введение

- Некоторые определения понятия «алгоритм»:
  - Хорошо определенная процедура, получающая некие данные как входные и преобразующая их в выходные данные
  - Последовательность вычислительных операций преобразующая входные данные в выходные
  - Способ решения конкретной (wellspecified) вычислительной проблемы
- Более общие определения:
  - Система последовательных операций (в соответствии с определёнными правилами) для решения какой-нибудь задачи (Теория алгоритмов)
  - Совокупность последовательных шагов, схема действий, приводящих к желаемому результату

Свойства алгоритма – 1/2

- Основные свойства алгоритма:
  - Правильность (correct)
  - Конечность (finite) – должен заканчиваться при определенных условиях или содержать конечное число шагов
  - Однозначность (unambiguous)
- Дополнительные свойства:
  - Модульность
    - В данном случае возможность использования «по частям»
  - Поддерживаемость (maintainability)
  - Функциональность
  - Устойчивость
    - Отсутствие или проверка внутри алгоритма «особых случаев»

Свойства алгоритма – 2/2

- Дополнительные свойства (продолжение) :
  - «Дружелюбность» (user-friendliness)
    - Например, простоту понимания (программистами)
  - Время на разработку (programmer time)
  - Простоту
  - Расширяемость
- Обычно выделяют «производительность» или «эффективность» – временн***ы***е или ресурсные затраты на выполнение
  - С некоторой точки зрения неэффективность показывает неприменимость алгоритма. Например, если алгоритм требует сотни лет вычислений он бесполезен или даже неверен

Анализ алгоритмов – 1/3

- Выделяют три основных вида анализа алгоритмов:
  - Оценка сверху: определяется функция (вид функции) $F_{max}(n)$, ограничивающая сверху объем ресурсов, требующийся для обработки исходных данных размером n
  - Средняя оценка: определяется функция $F_{avg}(n)$, задающая средний объем ресурсов для обработки исходных данных размером n, рассчитанный по всем вариантам исходных данных указанного объема. Для определения такой функции требуется сделать предположения о статистическом распределении данных в наборах исходных данных

Анализ алгоритмов – 2/3

- Основные виды анализа (продолжение):
  - Оценка снизу: определяется функция $F_{min}(n)$, задающая минимальный потребный для обработки исходных данных размером n объем ресурсов
- Эти оценки могут использоваться, например:
  - Оценка сверху – для определения требований к техническим средствам
  - Средняя оценка – для оценки средней производительности системы
  - Оценка снизу – для оценки минимального времени ответа системы

Анализ алгоритмов – 3/3

- Кроме указанных выше встречаются также:
  - Оценка максимального времени выполнения (amortized analysis) через максимальную длину последовательности операций
  - Конкурентный анализ, в котором определяется относительная эффективность алгоритма по сравнению с оптимальным алгоритмом (который может быть неизвестен) в той же категории (для тех же операций)