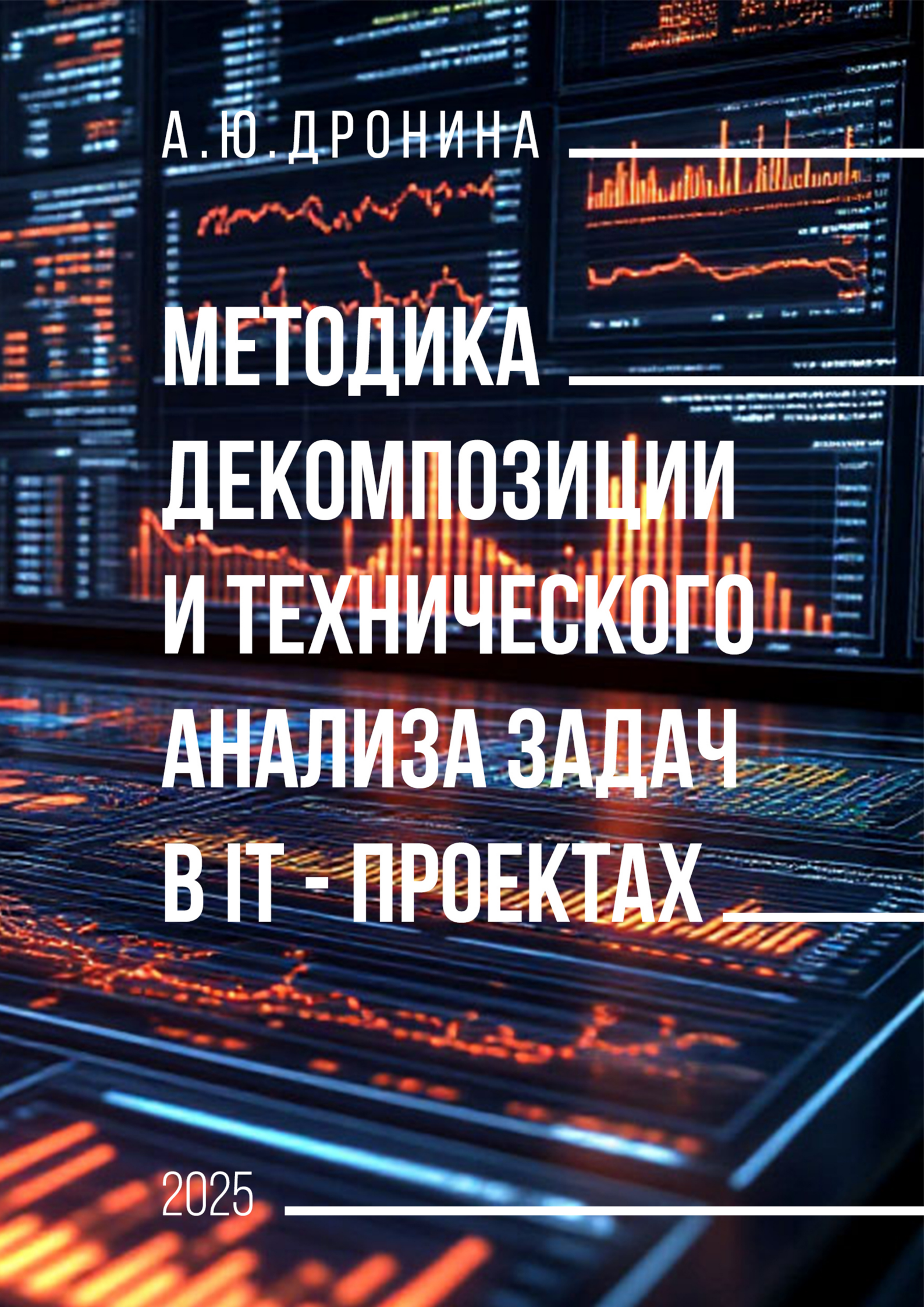

А.Ю.ДРОНИНА

МЕТОДИКА ДЕКОМПОЗИЦИИ И ТЕХНИЧЕСКОГО АНАЛИЗА ЗАДАЧ В IT - ПРОЕКТАХ

2025

А.Ю. Дронина

Методика декомпозиции и технического анализа задач в IT-проектах

Методическое пособие

Белгород
2025

ББК 65.29я73
Д 69

Дронина А.Ю.

Д 69 Методика декомпозиции и технического анализа задач в IT-проектах : методическое пособие / А.Ю. Дронина. – Белгород : ООО «Эпицентр», 2025. – 33 с.

ISBN 978-5-6054519-8-3

В работе представлена методика структурированной декомпозиции и технического анализа задач в IT-проектах. Она актуальна из-за роста сложности систем и необходимости точного планирования. Подход основан на разбиении задач на подзадачи, предварительном анализе требований и формировании итогового набора для реализации. Методика опирается на практики Agile и исследования, подтверждающие, что правильная декомпозиция повышает предсказуемость, качество кода и эффективность команд. Рассматриваются типы декомпозиции (функциональная, объектно-ориентированная, процессная и др.), рекомендации по теханализу и процесс от груминга до реализации. Практическая ценность подтверждается метриками и кейсами разработки продакшн приложений. Рекомендации полезны разработчикам и тимлидам: помогают снижать риски срыва сроков, накопления технического долга и улучшать планирование за счет глубокой проработки требований. Методика применима в проектах среднего и крупного масштаба.

ББК 65.29я73

ISBN 978-5-6054519-8-3

Содержание

Введение

В условиях современной разработки программных продуктов сложные задачи и проекты требуют тщательного планирования и анализа. Рост масштаба систем ведет к увеличению сложности реализации, наличию множества зависимостей и скрытых рисков [2]. Практика показывает, что ошибки на этапе планирования и анализа требований могут приводить к гораздо более серьезным проблемам в дальнейшем – их исправление обходится дороже, чем устранение дефектов на этапе эксплуатации [2]. Одной из ключевых техник, позволяющих повысить предсказуемость и успешность проектов, является декомпозиция задач. Декомпозиция подразумевает разбиение сложной задачи на более мелкие и понятные компоненты, которые легче анализировать, оценивать и реализовывать. Такой подход соответствует принципам управления сложностью в инженерии программного обеспечения, предлагая “разделять и властвовать” над сложной проблемой путем деления ее на части. Согласно руководствам по бизнес-аналитике, функциональная декомпозиция помогает управлять сложностью и уменьшать неопределенность, разбивая процессы или системы на более простые составляющие, которые можно анализировать независимо. Таким образом, декомпозиция служит инструментом снижения когнитивной нагрузки на команду разработки и более четкого понимания сути задачи.

Не менее важным аспектом успешного выполнения сложной задачи является проведение технического анализа (теханализа). Под техническим анализом в разработке подразумевается детальное исследование поставленной задачи до начала кодирования, с целью оценить все технические аспекты ее реализации: необходимые изменения в кодовой базе, новые модули, влияния на связанные компоненты, потенциальные риски и т.д. [2]. Результатом теханализа обычно становится документ или артефакт, описывающий план реализации функционала и полный перечень работ для различных участников (разработчиков, тестировщиков, аналитиков и др.) [2]. По сути, теханализ позволяет структурировать изначально разрозненные сведения о задаче и

перейти к четкому плану работ. Его применение предотвращает ситуацию, когда оценки трудозатрат даются неосознанно при поверхностном ознакомлении с требованиями, что характерно для неподготовленного груминга требований [2]. В контексте гибких методологий Agile подробный предварительный анализ сложной задачи зачастую является залогом ее успешного и своевременного выполнения [2].

Современные исследования подтверждают практическую значимость декомпозиции и теханализа. Например, согласно отчету Standish Group (Chaos Report), по состоянию на 2020 год значительная часть неудач проектов обусловлена проблемами на ранних этапах планирования требований [3]. Анализ 73 работ по оценке трудозатрат в Agile-проектах выявил, что несмотря на внедрение облегченных техник планирования, задача повышения точности оценок по-прежнему актуальна и напрямую связана с улучшением процессов декомпозиции требований и предварительного анализа [4]. Систематический обзор, опубликованный в IEEE Access, отмечает, что многие команды продолжают испытывать затруднения в разбиении пользовательских историй на задачи и в учете всех факторов, влияющих на сложность реализации [4]. Кроме того, исследования показывают, что размер задач существенно влияет на точность их оценки: слишком крупные задачи склонны к переоценке (разработчики закладывают избыточный запас времени), тогда как слишком мелкие задачи часто недооцениваются [5]. В одном из недавних экспериментов было установлено, что минимальная ошибка оценки достигается для задач средней продолжительности (порядка одного рабочего дня), тогда как задачи длительностью менее часа систематически требуют больше времени, чем ожидалось, а задачи более чем на неделю почти всегда переоценены [5].

Таким образом, актуальность разработки строгой методики декомпозиции и теханализа обусловлена потребностью IT-индустрии в инструментах повышения предсказуемости и качества программных проектов. В следующих разделах работы дается обзор основных подходов к декомпозиции задач (Глава 1), описываются цели и принципы проведения технического анализа (Глава 2), а

затем излагается пошаговая практическая методика применения данных подходов в рамках процесса разработки (Глава 3). Методика ориентирована на практическое использование в командах разработки для улучшения планирования спринтов, выявления скрытых зависимостей и распределения задач между исполнителями.

1. Теоретические основы декомпозиции задач

В управлении проектами и разработке ПО под *декомпозицией* понимается процесс разделения комплексной задачи, требования или проекта на более мелкие и управляемые части. Декомпозиция лежит в основе таких стандартных практик, как иерархическая структура работ (Work Breakdown Structure, WBS) в проектном менеджменте: согласно PMBOK, WBS представляет собой *“иерархическую декомпозицию работ, выполняемых командой проекта, на компоненты, обеспечивающие достижение целей проекта”* [6]. Иными словами, крупная цель разбивается на подцели, затем – на конкретные результаты и задачи, вплоть до элементарных действий. Применительно к разработке программного обеспечения, декомпозиция задач означает разбиение функциональности или требуемых изменений на отдельные шаги или подзадачи, каждая из которых может быть спланирована, реализована и протестирована относительно независимо. Формально декомпозиция задачи можно определить следующим образом: *это процесс разделения исходной задачи на несколько более простых задач, суммарная сложность которых не превышает сложности изначальной задачи*. Такое условие гарантирует, что разбиение не усложняет работу, а упрощает ее понимание.

Существует несколько подходов к декомпозиции задач в IT-проектах, в зависимости от того, какой признак лежит в основе разделения. В таблице 1 приведены основные виды декомпозиции и их сущность [1].

Таблица 1 – Основные виды декомпозиции задач в ИТ-проектах [1]

Вариант декомпозиции	**Сущность подхода и область применения**
Функциональная	Разбиение задачи по функциональным компонентам или модулям системы. Каждая подзадача соответствует отдельной функции или части функциональности продукта. Часто используется при проектировании архитектуры ПО и выделении модулей по функционалу.
Объектно-ориентированная	Декомпозиция проводится по объектам и сущностям предметной области. Задача разбивается на подзадачи, привязанные к

Вариант декомпозиции	Сущность подхода и область применения
	определенным объектам (классам) системы, учитывая их данные и методы. Подход применим в ООП-разработке для локализации изменений в пределах классов.
Процессная (по этапам)	Задача дробится по этапам или бизнес-процессам, необходимым для достижения результата. Подзадачи представляют собой последовательность шагов (workflow) выполнения исходной задачи. Такой подход востребован в управлении бизнес-процессами и последовательной разработке (waterfall, RUP).
Структурная (иерархическая)	Разбиение задачи на части согласно иерархической структуре системы. Например, крупное требование делится на под-требования, те – на еще более мелкие и т.д. Применяется в системной инженерии, когда строится иерархия требований или архитектурных компонентов.
Итеративная (по версиям)	Постепенная детализация решения через серию итераций. Задача сначала реализуется в упрощенном варианте, затем расширяется и уточняется на следующих циклах разработки. Широко используется в Agile-методологиях, позволяя получать результат постепенно (инкрементально).
Компонентная (модульная)	Декомпозиция по техническим компонентам системы. Задача делится на части, соответствующие независимым модулям, сервисам или компонентам, которые можно разрабатывать и поставлять отдельно, с последующей интеграцией. Полезна при микросервисной архитектуре и многомодульных проектах.

Следует отметить, что перечисленные виды не являются взаимоисключающими: на практике комбинируются различные подходы. Например, часто используется функционально-компонентная декомпозиция, когда система делится на модули по функциональности, а внутри модулей задачи делятся на этапы реализации. Выбор стратегии декомпозиции зависит от природы задачи и особенностей проекта. Главное – добиться такого разбиения, при котором каждая подзадача имеет четкую формулировку, ограниченный объем и может относительно независимо решаться. При этом сумма решений по подзадачам должна давать решение исходной задачи при минимальных накладных затратах на интеграцию результатов.

При этом правильная декомпозиция задач дает комплексный положительный эффект на процесс разработки и качество продукта [1]. В

таблице 2 обобщены ключевые преимущества декомпозиции, отмечаемые в литературе и практике.

Таблица 2 – Преимущества декомпозиции задач для процесса разработки [1, 5, 7, 8]

Преимущество	**Как проявляется при разработке**
Управление сложностью	Мелкие задачи проще понять и проанализировать. Разработчики лучше разбираются в требованиях каждой части, снижается риск упустить важные детали. В итоге общая сложность системы воспринимается через набор относительно простых элементов вместо одного «комбината».
Повышение качества кода	Работа над небольшими задачами позволяет сосредоточиться на деталях реализации. Снижается вероятность появления “быстрых грязных” решений: уделяется внимание качеству, покрытию тестами, рефакторингу каждой части. Также уменьшается технический долг, так как меньше неизученных областей остаётся “под ковром”.
Улучшение кода-ревью	Подзадачи влекут за собой меньшие изменения кода в каждом коммите. Небольшие диффы проще и быстрее просматривать коллегам. Исследования показали, что разбитые на небольшие части изменения приводят к меньшему числу ложных замечаний и большему числу предложений по улучшению, т.к. обозримый дифф легче анализировать тщательно. В целом, code review становится эффективнее и менее трудозатратным.
Возможность параллельной работы	Если крупную фичу разбить на независимые подзадачи, их можно распределить между несколькими разработчиками и тестировщиками. Таким образом, часть работ выполняется параллельно, сокращая суммарное время реализации. Параллельность особенно важна при жестких сроках: декомпозиция позволяет команде масштабироваться на задаче.
Повышение точности оценок	Мелкие задачи легче оценивать: разработчик лучше понимает объем конкретной подзадачи, меньше неопределенностей. Эстимейты по небольшим задачам более точны и стабильны. Как показал анализ, задачи длительностью ~1 день имеют минимальную ошибку оценки по сравнению с большими элементами. Совокупность точных оценок по подзадачам дает более надежную оценку всей фичи.
Ускорение планирования	В Agile-процессах (Scrum) команды планируют спринты исходя из емкости (capacity). Если фича не влезает в спринт целиком, декомпозиция решает проблему – отдельные части распределяются между спринтами. Таким образом, крупные элементы бэклога не блокируют планирование, а преобразуются в набор выполняемых итерационно задач.

Кроме перечисленного, декомпозиция тесно связана с такими преимуществами, как улучшение управления рисками (каждая мелкая задача несет меньше рисков и неопределенности, чем одна большая) и повышение прозрачности проекта для стейкхолдеров. Последнее особенно важно, так как детальный перечень задач, полученный при декомпозиции, позволяет всем участникам процесса видеть, что конкретно будет сделано для реализации требования. Это облегчает коммуникацию с заказчиком и внутри команды. Практики следят и за тем, чтобы при декомпозиции сохранялась трассируемость требований – возможность проследить связь между исходным бизнес-требованием и набором задач реализации [9]. В руководствах рекомендуют поддерживать такую трассируемость, например, в Agile-командах связывают high-level требования (эпики, фичи) с нижележащими user story и задачами, что позволяет убедиться – все запланированные подзадачи в совокупности закрывают исходное требование, и наоборот, каждая задача привязана к бизнес-ценности [9].

Также при разбиении задач важно учитывать ряд аспектов проекта. В частности, в кросс-функциональных командах могут существовать внешние зависимости – например, часть функциональности зависит от готовности компонентов, разрабатываемых другой командой (backend-сервисы, аналитические модули и пр.). Эти зависимости следует отражать отдельными задачами или пометками в плане (например, задача “дождаться API от команды X”) [1]. В многокомпонентных продуктах есть и внутренние зависимости между модулями, когда параллельная работа нескольких разработчиков над одной кодовой базой требует разбиения задач так, чтобы минимизировать конфликты изменений (merge conflicts) и блокировки друг друга [1]. Если полная изоляция невозможна, стоит определить порядок выполнения взаимосвязанных подзадач. Также следует заранее выявлять возможные блокеры – сложные места, неопределенности, спорные моменты архитектуры – и выносить работу по их проработке в отдельные технические задачи [1]. Наконец, не стоит забывать про требования по срокам: при жёстких дедлайнах выгодно декомпозировать задачу так, чтобы части можно было выполнять параллельно несколькими людьми, и

чтобы часть функционала могла быть протестирована и поставлена раньше остальной, если это необходимо бизнесу.

На рисунке 1 показана схема (mindmap), иллюстрирующая основные аспекты, которые рекомендуется учитывать при декомпозиции задачи. К ним относятся ограниченные сроки и параллельность работ, внешние и внутренние зависимости, общая сложность и неясности реализации, а также организация совместной работы команды. Держать в фокусе все эти факторы при разделении задачи – залог того, что полученная структура подзадач будет реалистичной и выполнимой в условиях проекта.

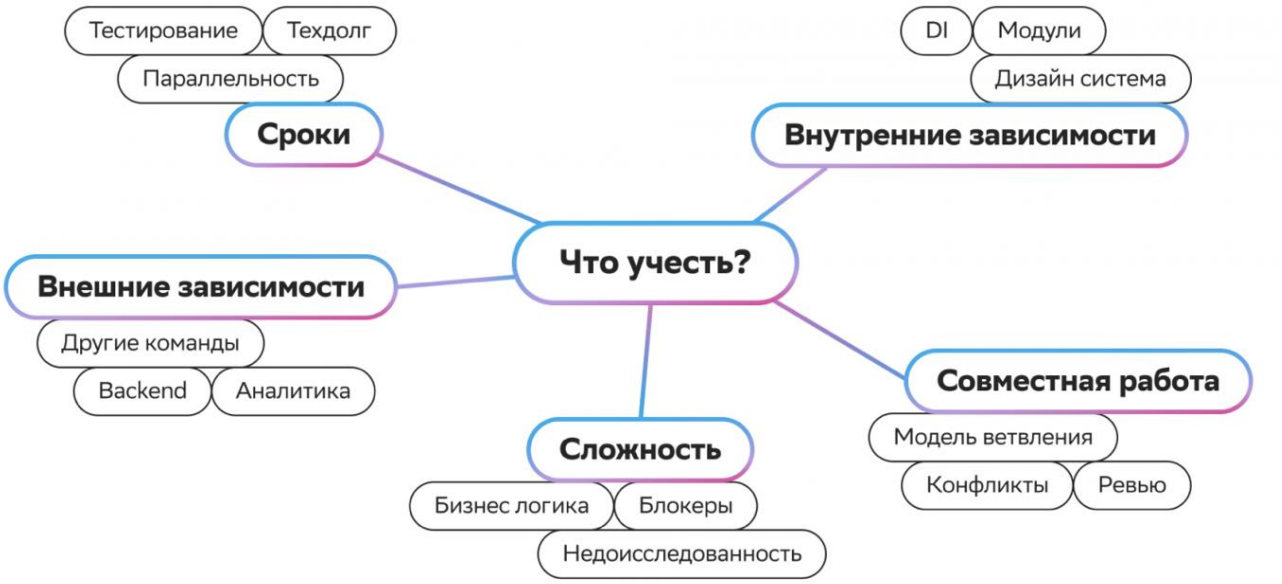

Рис. 1. Факторы, влияющие на разбиение задачи на подзадачи (схема) [1]. При декомпозиции следует учитывать временные ограничения (сроки, параллельность работы, техдолг и необходимое тестирование), внешние зависимости (готовность сторонних сервисов, команд, данных), внутренние зависимости (архитектура модулей, использование дизайна системы, общие компоненты, механизм DI и пр.), а также аспекты совместной разработки (принятую модель ветвления в VCS, возможные конфликты при слиянии, процесс код-ревью). Кроме того, оценивается общая сложность: наличие нетривиальной бизнес-логики, потенциальных блокеров, нехватка исследования по отдельным вопросам – все это может повлиять на структуру и количество подзадач

Подводя итог, декомпозиция задач – фундаментальная техника, позволяющая программистам и менеджерам прояснить структуру работ, необходимых для достижения цели, и тем самым облегчить управление проектом. Однако сама по себе декомпозиция не гарантирует успеха, так как необходимо еще правильно проанализировать каждую часть задачи и понять, как лучше ее реализовать. Этой цели служит технический анализ, которому посвящена следующая глава.

2. Технический анализ задач в разработке

Технический анализ (сокращенно *теханализ*) – это углубленное исследование задачи, выполняемое до (или в начале) этапа ее реализации. Основная цель теханализа – ответить на вопрос *“как именно будет реализована данная функциональность?”*, выявив при этом все необходимые изменения в системе, затрагиваемые модули, потенциальные проблемы и варианты решений [2]. В результате проведения теханализа формируется артефакт (технический документ, отчет или протокол анализа), который содержит подробное описание всех этапов разработки функционала либо анализа возникшей проблемы. Такой документ служит, во-первых, основой для декомпозиции задачи – именно на базе выявленных этапов и изменений формируются конкретные задачи для разработчиков, тестировщиков, девопс-инженеров и т.д. [2]. Во-вторых, результаты теханализа обеспечивают унификацию понимания задачи между всеми участниками, так как документ обсуждается с постановщиком задачи (бизнес-заказчиком или аналитиком), командой разработки и ответственным тимлидом, что гарантирует согласование плана работ и выявление ошибок понимания требований еще до написания кода [2].

Следует подчеркнуть отличие теханализа от других видов аналитических работ. Если бизнес-анализ фокусируется на *требованиях* и *бизнес-ценности* функции, то технический анализ концентрируется на *внутреннем устройстве и шагах реализации*. По сути, теханализ – это переходный этап между постановкой задачи («что нужно сделать») и ее непосредственным выполнением («как мы это сделаем»). В Agile-разработке зачастую теханализ проводится для задач, которые на груминге получили высокую оценку сложности или вызвали разногласия в оценках у разных разработчиков [2]. Его также привлекают в нестандартных ситуациях – например, при расследовании сложного дефекта, когда непонятна причина проблемы и требуется исследование системы для ее обнаружения [2]. В любом случае, решение о необходимости теханализа диктуется здравым смыслом и опытом команды, так как программисты должны

четко понимать, когда без предварительного подробного разбора задачи не обойтись, чтобы избежать авралов, переработок и хаотичного рефакторинга кода позже [2].

Хотя технический анализ достаточно гибкая по форме практика. Тем не менее, можно выделить общие принципы, которыми стоит руководствоваться при его проведении:

- *Достаточная, но не избыточная детализация.* Главный вопрос – какой глубины должен быть теханализ. С одной стороны, недостаточно подробный анализ может пропустить важные детали, и тогда цели не будут достигнуты. С другой стороны, чрезмерно подробный, «низкоуровневый» теханализ с расписыванием каждого шага реализации вплоть до псевдокода – неэффективен и тратит время [2]. Практика показывает, что оптимальным является верхнеуровневое описание, когда теханализ должен перечислить, *что* и *где* нужно изменить или разработать, без углубления в мелочи реализации, которые программист сможет продумать уже при кодировании [2]. Например, если необходим новый модуль – указать, какой и куда его интегрировать, а не писать полный листинг кода. Если изменяется база данных – описать, какие таблицы или поля потребуется добавить, а не приводить полный скрипт миграции. Такой баланс позволяет получить ценность от анализа и не превратить его в самодостаточный “мини-проект”.
- *Проверка существующего кода и функционала.* Выполняя теханализ, разработчик изучает текущую реализацию и возможно, требуемая функциональность частично уже реализована или похожие решения есть в других модулях. Важно отыскать, что можно переиспользовать, какие места системы затронет новое изменение. Это часто включает чтение кода, диаграмм, документации. Например, при добавлении нового фильтра товаров на сайте нужно проверить, какие уже есть механизмы фильтрации, и можно ли встроиться в них. Такой аудит кода и компонентов – ключевая часть теханализа, позволяющая избежать дублирования и понять зону влияния изменений.

- *Фиксация результатов и предположений.* Весь процесс размышления в теханализе желательно отражать письменно. Документ теханализа обычно содержит: описание исходной постановки; обсуждение различных подходов к реализации (если есть варианты); перечисление необходимых изменений с указанием модулей/классов/сервисов; возможные риски и вопросы; оценку трудозатрат по каждому подпункту. Даже если некоторые моменты очевидны исполнителю, их стоит записать – это проверяется коллегами на ревью теханализа. Документ также служит затем “дорожной картой” при выполнении работ. Поэтому теханализ по сути представляет собой план действий, написанный в свободной форме, но ясно структурированный по шагам (подзадачам).

- *Коллаборация и ревью.* Теханализ – не личное дело одного разработчика. После того, как исполнитель подготовил черновик технического плана, его направляют на проверку другим членам команды и заинтересованным лицам [2]. Часто в обсуждении участвуют тимлид или архитектор, автор бизнес-требования (аналитик/продукт-менеджер) и другие разработчики, особенно если затрагиваются их зоны в коде. Цель ревью – убедиться, что предложенное решение оптимально, ничего не упущено, а также все согласны с объемом работ. Такое коллективное обсуждение приносит пользу и по статистике, обзор теханализа позволяет улучшить план или найти ошибки в понимании задачи, предотвращая их перенос в реализацию [2]. После учёта замечаний теханализ утверждается, и команда получает “единый источник правды” о том, как будет делаться задача. Отметим, что в Agile-командах подобный процесс часто проходит неформально – в виде обсуждения на командной встрече, – но суть остается: теханализ должен быть проверен и согласован.

При этом унифицированной строгой структуры документа теханализа нет – каждый проект вырабатывает свой шаблон. Тем не менее, можно очертить типичное содержание такого документа.

Для начала важно понять краткое описание задачи. Что требуется от системы, какие новые функции или изменения нужны (пересказ постановки задачи своими словами разработчика, чтобы зафиксировать понимание).

Далее какие модули, компоненты, страницы, сервисы системы затрагивает данная задача. Здесь же – что не входит в ее границы (например, “бэкенд-сервис X не изменяется, предполагается использование его существующих API”).

После идет описание того, как сейчас работает соответствующий функционал (если речь о доработке) или какие аналогичные механизмы есть. Сюда же – результаты исследования кода: ключевые классы, функции, конфигурации, связанные с задачей. Например: “В модуле OrderService есть класс OrderFilter, сейчас поддерживает фильтрацию по цене и дате; нужно добавить фильтр по популярности – возможно, расширив этот класс”.

Следующим идет проект предлагаемого решения. Этот раздел – ядро теханализа. Он разбивается на подпункты по каждому изменению или компоненту:

- новые сущности или модули, которые нужно создать (с указанием, где они будут находиться, как взаимодействовать с существующими частями);
- изменения в существующем коде (например: “изменить метод A класса B, чтобы поддерживал новый параметр; добавить enum со значением таким-то”);
- изменения в базе данных (создать новую таблицу, изменить колонку и т.д., с указанием структуры);
- изменения во внешних интерфейсах (REST API, сообщения очередей и пр., если необходимо);
- требования к конфигурации, инфраструктуре, безопасности, если есть (например, “потребуется открыть новый порт”, или “нужен новый ключ в файле настроек”);
- *варианты реализации*: если есть альтернативные пути достижения цели, их плюсы/минусы и обоснование выбора.

Далее описывается, как планируется проверять выполненную задачу: какие случаи важно протестировать, нужны ли нагрузки, какие модули потребуют модульного тестирования или обновления автотестов. Это важно отметить, чтобы не забыть включить оценку тестирования в общую трудоемкость.

После перечисления всех изменений указывается примерная оценка по времени (или в сторипойнтах) для каждого пункта и суммарно. Эта оценка часто уточняет изначальную грубую прикидку, данную на груминге.

В завершение фиксируются выявленные риски (например: “Есть риск, что внешнее API изменится, тогда придется доработать коннектор”) и допущения (assumptions), которые были положены в основу анализа (“Предполагается, что библиотека X будет обновлена до версии Y; исходные данные приходят в правильном формате; пользователь имеет права администратора”, и т.д.). Явное указание допущений помогает затем проверить их и избежать неожиданностей.

Стоит отметить, что объем теханализа зависит от сложности задачи. Для небольшой доработки на пару дней теханализ может уместиться в одну страницу текста, тогда как для новой подсистемы на несколько месяцев – это может быть и десятки страниц с диаграммами. Однако даже в последнем случае стараются не детализировать абсолютно всё, а оставлять свободу для agile-подхода при реализации. Главное – теханализ должен разобрать “скелет” задачи и показать, какие “органы” потребуются, не расписавая “каждую клеточку”. Как образно отмечает Евгений Шалаев, чрезмерно подробный теханализ сродни хождению по битому стеклу – болезненному и бесполезному занятию, отвлекающему от собственно программирования [2].

Для иллюстрации рассмотрим упрощенный пример. Допустим, стоит задача: *“Реализовать новую панель фильтров товаров на веб-сайте: на десктопе – две кнопки (сортировка по цене и по популярности), на мобильном – четыре кнопки; использовать предоставленные макеты UI”*. На груминге команда понимает, что задача нетривиальная, оценки разнятся – решено сделать теханализ. Разработчик изучает текущий модуль каталога товаров и

обнаруживает, что логика сортировки на бэкенде уже есть (по цене, по популярности), но на фронтенде нужно добавить UI и вызывать соответствующие API. Также находит, что для модальных окон поиска используется общий компонент, но в макете фильтров нужен иной дизайн – вероятно, придется сделать новый компонент. Он оформляет теханализ примерно так:

- Создать новый React-компонент FiltersPanel: на десктопе отображает 2 кнопки (SortByPrice, SortByPopularity), на мобильной версии – 4 (две упомянутые + кнопка открытия модального окна поиска + кнопка дополнительных фильтров по весу). Использовать существующий UI-кит для кнопок, кроме случая модального окна поиска – там нужен новый дизайн.
- В состояниях (Redux store) добавить поле currentSort с возможными значениями (NONE, BY_PRICE, BY_POPULARITY и т.д.).
- В бэкенд API /products убедиться, что параметры сортировки принимаются – да, есть параметр sortType. Добавить его передачу при запросе с фронта.
- Написать обработчики нажатия кнопок, меняющие состояние сортировки и инициирующие перезапрос списка товаров.
- Для “дополнительных фильтров по весу” – требуется новая функциональность: в модальном окне выбрать диапазон веса. Предлагается реализовать новый компонент WeightFilterModal со своими состояниями. Потребуется доработка бэкенда: добавить параметр weightRange в запрос товаров и поддержку его в сервисе фильтрации. Это выделяем отдельной подзадачей для бэкенда.
- Оценка: фронтенд – 3 дня, бэкенд – 1 день, тестирование – 1 день, всего ~5 дней.
- Риски/вопросы: нужно уточнить у аналитика, должны ли фильтры сохраняться при переходе между страницами (если да – добавить сохранение в URL или localStorage). Также вопрос: должна ли вторая кнопка (доп. фильтры)

отличаться по стилю – в макете она шире; возможно, UI-кит придется дорабатывать.

Этот пример демонстрирует логику, когда разработчик разбил задачу на конкретные пункты, понял, где что менять, и отразил это текстом. Получив такой документ, команда и заказчик смогли бы согласовать решение, поправить неточности (например, уточнить про UX поведение кнопок) до того, как начнется кодирование.

Однако несмотря на очевидные плюсы теханализа, важно помнить, что *он не нужен для каждой задачи*. Простые, хорошо понятные задачи в Agile-проектах зачастую можно реализовать сразу, придерживаясь принципа YAGNI (You Ain't Gonna Need It) – не тратя время на излишние предварительные исследования. Технический анализ – это инвестиция времени разработчика; она окупается, если предотвращает более серьезные потери времени из-за переделок и ошибок. Но если задача тривиальна или время критически ограничено, команда осознанно может принять решение действовать сразу, без полного анализа, чтобы не затягивать доставку. Такой подход приемлем "на свой страх и риск" в определенных случаях, однако нужно понимать, что это увеличивает вероятность возврата к задаче позже для исправления последствий [2]. Оптимально – наработать чувство баланса, когда теханализ применяется там, где без него действительно не обойтись (например, новые большие функции, сложные интеграции, архитектурные изменения), но не тормозит команду в очевидных ситуациях.

Подводя итог можно отметить, что технический анализ является важной частью методики декомпозиции больших задач. Он обеспечивает обоснованность и целостность разбиения задачи на подзадачи, служит связующим звеном между бизнес-требованиями и планом реализации, а также инструментом командной коммуникации. В следующей главе мы рассмотрим практическое применение этих принципов – как именно в реальном процессе разработки происходит декомпозиция задачи с этапом теханализа, и какие шаги следует выполнить команде, чтобы внедрить данную методику в работу.

3. Практическая методика декомпозиции задачи (с этапом теханализа)

В данной главе представлена поэтапная методика применения декомпозиции и технического анализа на практике, основанная на реальном сценарии разработки мобильного приложения [1]. Предположим, что команде разработки поступила новая фича (требование) от бизнеса. Необходимо пройти через несколько шагов – от первоначального ознакомления с требованием до полного набора задач в трекере, готовых к выполнению. На рисунке 2 схематично показаны основные этапы этого процесса.

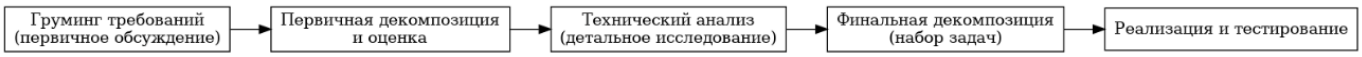

Рис. 2. Этапы анализа и декомпозиции задачи в процессе разработки (блок-схема). После первичного обсуждения требования на груминге следует выполнить предварительную декомпозицию и дать грубую оценку. Далее проводится углубленный технический анализ, по результатам которого формируется финальный список подзадач (финальная декомпозиция с уточненными оценками). Завершается процесс непосредственно реализацией и тестированием всех полученных задач. Представленная последовательность шагов интегрируется в Scrum-процесс планирования спринтов

Ниже подробнее рассматриваются ключевые шаги методики: 3.1 Первичная декомпозиция и оценка, 3.2 Проведение технического анализа, 3.3 Финальная декомпозиция и планирование выполнения.

3.1 Первичная декомпозиция и оценка задачи

Первичный этап начинается сразу после того, как новая задача (фича) поступает в команду. В Scrum-командах практикуется груминг (refinement) бэклога – встреча, на которой команда разработки совместно с аналитиком и владельцем продукта обсуждает требование. Цель – понять суть задачи и

прикинуть ее сложность. Как отмечалось ранее, зачастую уже на этом этапе понятно, что задача крупная и может занять более 1–2 дней работы [1]. В таких случаях разработчики приступают к предварительной декомпозиции прямо во время или сразу после груминга, чтобы дать более обоснованную оценку, а не “пальцем в небо” [2].

Предварительная декомпозиция проводится примерно, без глубокой проработки – зачастую на листке бумаги, доске или в черновике задачи. Разработчик, ответственный за оценку, разбивает фичу на ряд укрупненных подпунктов работ. Как правило, список типовых подзадач для большинства пользовательских историй известен заранее (на основе прошлых схожих задач). Например, разработчик мобильного приложения может использовать такой шаблонный список подзадач для новой фичи [1]:

- Технический анализ фичи. Провести анализ и оформить документ (о котором говорилось в главе 2). Результат – уточненный план и финальная декомпозиция.
- Доработка существующих сущностей. Внести изменения в уже существующий код: модифицировать модели данных, расширить API, поправить бизнес-логику, изменить UI-элементы, связанные с новой функцией.
- Реализация новой логики. Написать новый код, отвечающий за функциональность фичи: новые классы, модули, экраны, запросы к серверу и т.п.
- Работа с зависимостями (интеграция модулей). Если проект многомодульный, возможно потребуется перемещение или создание компонентов в разных модулях, изменение интерфейсов между модулями. Здесь учитывается архитектура приложения – например, выделение новых интерфейсов, чтобы не было тесной связи между модулями [1]. Кроме того, проверяется необходимость обновления версий библиотек или взаимодействия с другими командами (если их компоненты участвуют).
- Написание тестов. Оценить объем модульных и интеграционных тестов, которые нужно подготовить для новой логики. В современных командах

принято покрывать основную бизнес-логику unit-тестами, что обязательно закладывается во время [1].

- Отладка и самопроверка. Зарезервировать время на самостоятельное тестирование разработчиком и исправление обнаруженных багов перед передачей задачи в QA.

Такой перечень носит ориентировочный характер. Конкретно в каждой задаче какие-то пункты могут отсутствовать (например, не нужно создавать новую логику, если функциональность достигается лишь изменением параметров существующей), или добавляться дополнительные (например, “Настроить feature-флаг для включения/выключения новой функции”). Однако базовый шаблон помогает ничего не упустить. На рисунке 3 представлен пример визуализации первичного разбиения фичи на составляющие задачи.

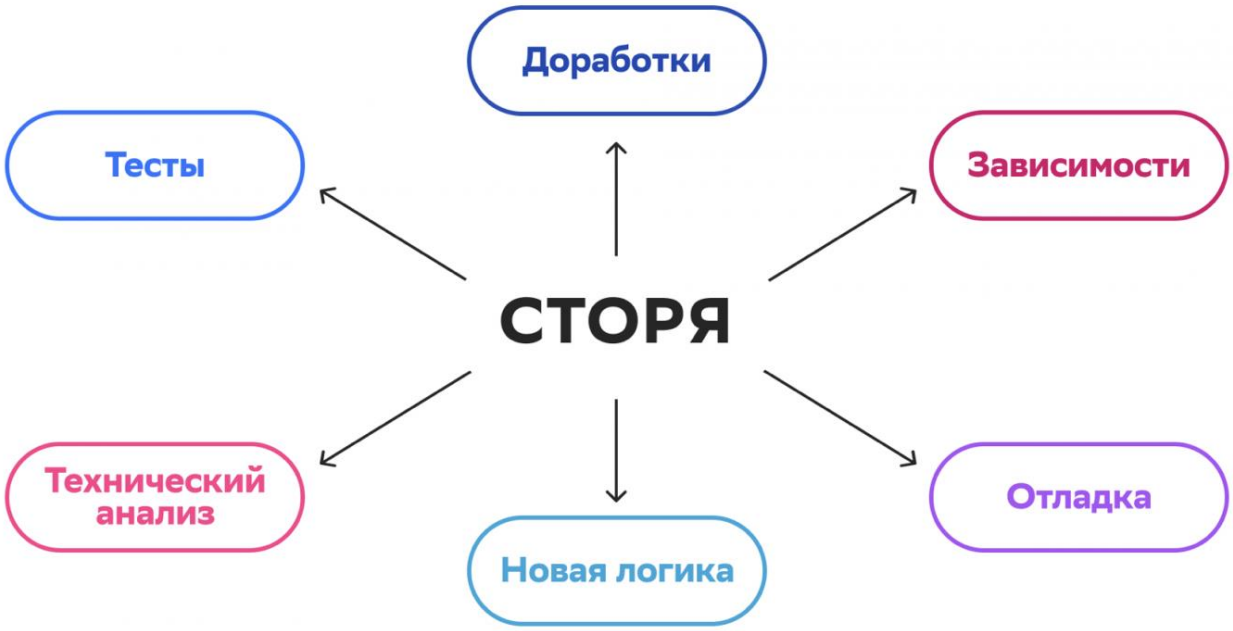

Рис. 3. Схема первичной декомпозиции задачи на типовые компоненты. В центре находится “Юзер-стори” (фича), вокруг нее блоки основных категорий работ: технический анализ, доработки существующего кода, новая логика, работа с зависимостями, тесты, отладка. Данный набор отражает стандартные аспекты, которые предстоит выполнить при реализации крупной функциональности [1]. На этапе первичной декомпозиции определяется, затрагиваются ли эти аспекты в текущей задаче, и оценивается трудоемкость по каждой категории

Имея перед глазами разложенную на 5–7 компонентов задачу, команда может прикинуть трудоемкость каждого компонента (например, “теханализ – 0.5 дня, доработки – 1 день, новая логика – 2 дня, интеграция – 1 день, тесты – 0.5 дня, отладка – 0.5 дня, итого ~5.5 дней”). Таким образом рождается первичная оценка, которая сообщается бизнесу или используется для планирования спринта. При этом рекомендуется добавлять небольшой резерв на риски – например, +20% к суммарной оценке, – учитывая, что детальный анализ еще не проводился и могут всплыть новые задачи [1]. Задачу целесообразно сразу пометить как требующую теханализа и декомпозиции: это сигнал для команды, что в начале следующего спринта нужно заложить время на теханализ.

Важно подчеркнуть, что первичная декомпозиция носит черновой характер. Она не обязательно фиксируется где-то формально (разве что в комментариях к задаче), и служит текущим потребностям оценки. Однако качество этого шага влияет на точность оценок, как отмечалось во введении, чем лучше разбиение – тем ближе команда попадет в оценку. Наоборот, если ограничиться одной цифрой “на ощущении”, есть риск сильного расхождения с реальностью, что чревато срывом сроков или авралом.

3.2 Проведение технического анализа

После того как задача отобрана в спринт или поставлена в план, первым шагом ее реализации становится технический анализ (если было решено его делать). На практике это означает, что разработчик (или несколько, если задача большая) тратит определенное время – обычно от нескольких часов до нескольких дней – *исключительно на аналитику и планирование*, не переходя к написанию кода. Как отмечалось в главе 2, на этом этапе изучаются требования и код, продумывается архитектура решения, после чего результаты оформляются документально и проходят ревью.

Например, в компании описываемый процесс может быть встроен в цикл разработки так: сначала груминг и оценка, потом теханализ, затем уточнение оценки и разбивка на задачи, и лишь потом – непосредственно кодинг [1]. В среднем на технический анализ крупной фичи уходит порядка 1–2 дней, однако

эти затраты окупаются сокращением времени основной разработки за счет уменьшения переделок и доработок.

На рисунке 4 показан условный результат теханализа для некоторой задачи в виде схемы декомпозиции. Этот рисунок отражает, как из исходных крупных блоков (описанных на этапе 3.1) получаются более детализированные подзадачи. Видно, что каждый аспект раскрылся более подробно: например, “доработки” разделились на несколько конкретных изменений (бизнес-логика, API, модель данных, UX-правки), новая логика – на создание UI-элементов и новых моделей, и т.д. Именно на основании этой детальной схемы команда формирует окончательные задачи в системе управления проектом.

Рис. 4. Результат проведения теханализа – детализация компонентов задачи. Центром по-прежнему является исходная “стория” (задача), вокруг которой сгруппированы области работ: технический анализ, доработки, новая логика, зависимости, отладка, тесты [1]. В ходе теханализа каждая область была разобрана на конкретные шаги. Например, технический анализ включал коммуникации с аналитиком и дизайнером, анализ кода, описание изменений и рисование схем; “доработки” распались на обновление бизнес-логики, изменение API, расширение модели данных и UX-изменения; “зависимости” – на выделение новых модулей, рефакторинг интерфейсов; “отладка” – на настройку окружения, взаимодействие с QA и исправление багов; и т.д. Такая проработка позволила получить полный список необходимых задач

Следует обратить внимание, теханализ может выявить, что для некоторых крупных частей требуется еще более мелкая декомпозиция. В практике часто применяется правило, что если подзадача все еще слишком объемна (например, “Изменить бизнес-логику Y” затрагивает множество классов и может выполняться разными людьми), ее нужно дальше разделить. Это называют вторым уровнем декомпозиции. В итоге, декомпозировать имеет смысл до тех пор, пока задачи не станут достаточно атомарными – обычно такими считают задачи, которые можно выполнить одному человеку за 1–2 дня, не распадающиеся на самостоятельные части [1]. Если попытаться дробить бесконечно (например, каждую строчку кода выделять в задачу) – эффект будет отрицательный, управлять станет сложнее, поэтому нужна мера.

В ходе теханализа первоначальные оценки трудозатрат могут уточняться. Нередко после глубокого погружения становится ясно, что какие-то части задачи проще или сложнее, чем казалось. Тогда разработчик обновляет свою оценку и сообщает новую цифру менеджеру. Например, первично оценивалось 5 дней, а после теханализа видно, что потребуется 6 или наоборот 4 – эта информация ценна для планирования релиза. В рассматриваемой методике предполагается, что на момент начала выполнения теханализа задача уже включена в спринт, поэтому уточнение оценки влияет скорее на *объем взятых обязательств* (если обнаружилось больше работы, возможно, часть пойдет в следующий спринт) или на *перераспределение ресурсов* (подключение дополнительных разработчиков).

Итак, выходом этапа теханализа является:

1. Документ с описанием реализации. Он может храниться в wiki, в файле или прямо в таск-трекере в комментариях – в зависимости от принятых процессов. Главное, чтобы его можно было читать и комментировать всем заинтересованным.

2. Обновленная декомпозиция задачи на конкретные подзадачи. Часто оформляется в виде чек-листа или подзадач (sub-tasks) в системе управления проектами (Jira, Trello, Azure Boards и т.д.). Каждая подзадача имеет описание,

оценку и ответственного (поначалу может быть «to do», потом назначается исполнителю).

На этом шаге методики команда получает полную ясность: *что именно нужно сделать* для закрытия исходной задачи. Далее остается приступить к непосредственному выполнению – но делается это уже не вслепую, а с четким планом.

3.3 Финальная декомпозиция и планирование выполнения

Финальная декомпозиция представляет собой список конкретных задач, готовых к постановке в работу. Если предыдущие шаги были выполнены тщательно, то эти задачи:

- взаимно независимы (насколько возможно) – то есть их можно делать в параллель или любом разумном порядке без конфликтов;
- имеют понятные критерии завершения (Definition of Done) и могут быть покрыты тестами отдельно;
- описаны понятным языком и согласованы командой.

При формировании финальной декомпозиции следует группировать работы так, чтобы минимизировать пересечения. Например, разумно объединить связанные изменения в одну задачу, чтобы их делал один человек, и избежать ситуации, где два разработчика меняют один и тот же модуль по разным задачам и мешают друг другу. Хорошая практика – слабосвязные группы задач: каждая группа решает свою под-проблему, и между группами по возможности мало точек взаимодействия [1]. Например, можно выделить подзадачу “Реализация фронтенд-части фильтров” и “Доработка бэкенд-API для фильтров” – их разные исполнители могут делать параллельно, стыкуясь через чётко определенный контракт.

Продолжая пример с панелью фильтров: после теханализа список задач мог выглядеть так:

1. Теханализ и дизайн решения (завершен).

2. Фронтенд: Добавить панель фильтров на страницу каталога. Описание: создать компонент FiltersPanel с 2 (desktop) / 4 (mobile) кнопками, реализовать переключение состояния сортировки, интегрировать с уже существующим списком товаров. Оценка: 3 дня. Ответственный: Dev A.

3. Фронтенд: Реализовать модальное окно дополнительных фильтров. Описание: компонент WeightFilterModal, открыть по кнопке “Еще фильтры”, позволить выбрать диапазон веса, при применении – фильтровать список. Оценка: 2 дня. Ответственный: Dev B.

4. Бэкенд: Доработать метод получения товаров. Описание: в сервисе ProductService добавить поддержку параметра weightRange, обновить SQL-запрос; убедиться, что сортировка по популярности учитывает новые данные. Оценка: 1 день. Ответственный: Dev C.

5. Обновить автотесты и написать новые. Описание: фронтенд-юнит-тесты на новый компонент FiltersPanel, интеграционный тест на цепочку фильтрации, бэкенд-тест на новую логику weightRange. Оценка: 1 день. Ответственные: Dev A + QA.

6. Регресс-тестирование и отладка. Описание: собрать сборку, проверить в разных браузерах, на разных устройствах; исправить выявленные баги UI. Оценка: 1 день. Ответственные: Dev A + QA.

Естественно, этот перечень может оформляться по-разному. В Scrum-командах часто не заводят отдельные задачи на тестирование или отладку – эти активности входят в Definition of Done задач разработки. Здесь они выделены для полноты картины. Важнее, что подобный финальный скоуп задач полностью покрывает все работы, необходимые для реализации исходной фичи [1]. В дальнейшем, при отслеживании статуса, менеджер видит прогресс по каждой подзадаче, а по выполнению всех – может с уверенностью сказать, что фича готова.

На рисунке 5 приведен пример финальной декомпозиции на схеме: условная большая задача “Реализовать N” разбита на конкретные подпроцессы –

добавить ресурсы X, написать модуль Y, исправить логику Z и т.д. Это соответствует тому списку, который будет реализован.

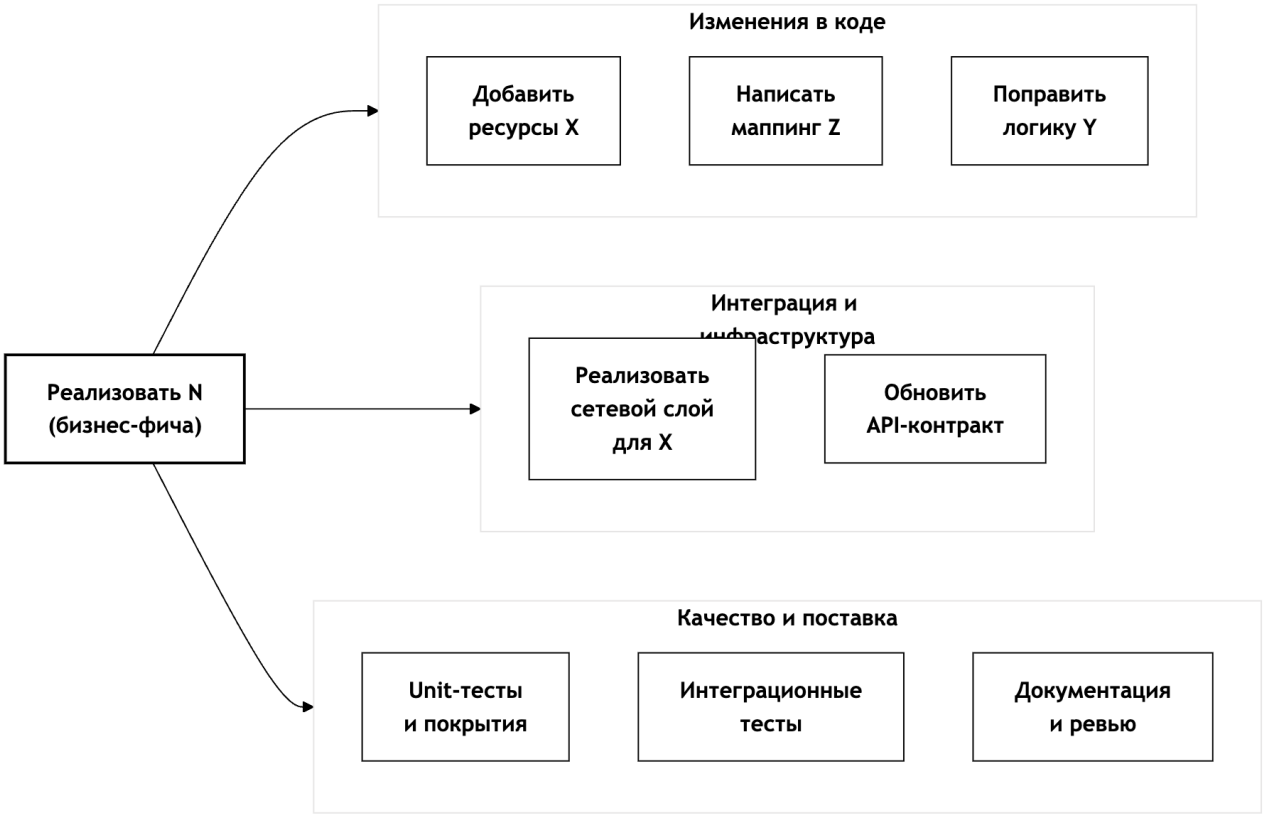

Рис. 5. Пример финальной декомпозиции: задача "Реализовать N" делится на набор конкретных подзадач: "Добавить ресурсы X", "Написать маппинг Z", "Реализовать сетевой слой для X", "Поправить логику Y" и т.п. (пример из практики). Такие подзадачи формируются на основании теханализа и представляют собой атомарные единицы работы, не требующие дальнейшего дробления

Финальная декомпозиция обычно завершается планированием исполнения: распределением задач между исполнителями (если это не было решено ранее) и определением очередности. Если команда небольшая, некоторые разработчики могут брать несколько подзадач по порядку. Важно убедиться, что порядок работ учтен: например, сначала желательно выполнить задачу по доработке API, а потом – фронтенд, который этим API пользуется (чтобы тестирование фронта проходило уже с рабочим сервером). В Scrum это

делается на планировании спринта: команда берет в работу весь полученный скоуп подзадач, оценивает суммарно и включает в спринт.

После этого начинается реализация. В процессе разработки могут обнаружиться новые детали, но благодаря проведенным шагам их число минимально. Если же какие-то мелкие дополнительные задачи всплывают, их обычно либо быстро решают в рамках существующих (не меняя больших планов), либо заводят новые задачи и тоже выполняют. Методика гибкая: всегда можно скорректировать план, добавив или убрав что-то, если изменилась ситуация.

По завершении всех работ, новые функциональности проходит тестирование, и задача считается выполненной. Команда проводит демо, убеждается, что все части успешно интегрированы и бизнес-требование удовлетворено. Отдельно можно отметить – методика декомпозиции облегчает сопровождение и анализ проекта после: наличие в таск-трекере множества мелких задач вместо одной большой позволяет в будущем проследить историю изменений, быстро найти, где и кем реализована та или иная часть функциональности, а также точечно оценить влияние при регресс-тестировании.

Заключение

В данной работе представлена академически обоснованная методика декомпозиции задач и проведения технического анализа в IT-проектах. Она основана на комбинации теоретических принципов и практического опыта. Ключевые выводы и рекомендации можно сформулировать следующим образом:

- Декомпозиция задач является необходимым инструментом управления сложностью при разработке программного обеспечения. Разделяя крупные задачи на мелкие, команда снижает когнитивную нагрузку, лучше понимает требования и снижает риски пропустить важные детали. Практика показывает, что такой подход универсален и применим в проектах разного масштаба – особенно он оправдан для сложных и длительных разработок, где “разделение слона на кусочки” позволяет планомерно достичь цели.
- Технический анализ дополняет декомпозицию, обеспечивая качество планирования. Если декомпозиция отвечает на вопрос “на какие части разбить задачу?”, то теханализ – “что нужно сделать в каждой части?”. Выполнение теханализа перед реализацией крупной задачи доказало свою эффективность: по сравнению с “сразу кодить”, теханализ позволяет выявить проблемы на ранней стадии, синхронизировать понимание внутри команды и избежать значительных переделок [2]. Хотя теханализ требует временных затрат, он окупается за счет более точных оценок и экономии времени на исправлении ошибок.
- Комплексное применение декомпозиции и теханализа улучшает ключевые показатели разработки. В работе рассмотрены преимущества: повышение качества кода (благодаря фокусировке на небольших участках и лучшему code review), ускорение обзора кода (меньшие change-set’ы легче ревьюить, что подтверждается исследованиями (di Biase et al., 2019)), параллельная работа (сокращение time-to-market, что особо ценно при жестких дедлайнах), точность оценок (минимизация отклонений за счет детализации –

см. рис. 1) и более эффективное планирование спринтов (задачи укладываются в итерации). В совокупности это ведет к повышению вероятности своевременной поставки функционала без снижения качества.

- Методика декомпозиции не является жесткой инструкцией, а набором принципов. Как и любой метод управления, ее следует адаптировать под контекст проекта. Для небольших задач избыточная декомпозиция может быть не нужна – важно применять методику там, где она действительно приносит пользу. Опыт показывает, что для крупных и комплексных задач предложенный подход универсален, тогда как для мелких user story в Scrum избыточный формализм может замедлять команду. Поэтому руководителю разработки важно прививать команде гибкость в использовании метода, обучать определять границу применимости теханализа.
- Внедрение методики требует организационных усилий и дисциплины команды. Необходимо, чтобы все участники – разработчики, аналитики, тестировщики – понимали ценность предварительного анализа и участвовали в нем. Полезно документировать успешные кейсы: например, сравнить время, ушедшее на задачу с теханализом и без – зачастую такие сравнительные примеры убеждают команду в эффективности подхода (Pasuksmit et al., 2024). Важна поддержка со стороны менеджмента, так как время на теханализ должно закладываться в планы, а не считаться “потерянным”.

В заключение отметим, что предложенная методика отражает современные тенденции инженерной практики: стремление к прозрачности разработки, управляемости процессов и снижению неопределенности. Декомпозиция и технический анализ делают процесс разработки более предсказуемым, что особенно ценно в условиях быстро меняющихся требований и высокой сложности современных IT-систем. Как показал опыт передовых компаний и исследований, применение таких подходов позволяет существенно повысить успешность проектов. Тем не менее, методика не претендует на звание “серебряной пули” – она требует грамотного применения и здравого смысла. В руках опытной команды декомпозиция и теханализ превращаются в мощные

инструменты, тогда как слепое следование им без понимания может снижать гибкость. Поэтому автор рекомендует читателям адаптировать изложенные рекомендации под свои проекты, экспериментировать, накапливать собственный опыт – и тем самым вносить вклад в дальнейшее развитие культуры инженерного анализа и планирования в сфере разработки программного обеспечения.

Список использованных источников

1. Dronina, A. Декомпозиция задач: как разработчику съесть слона? Habr. – 2024. [Электронный ресурс] – Режим доступа: https://habr.com/ru/companies/docdoc/articles/886460/

2. Shalaev, E. Die But Do: теханализ и почему без него разработка обречена на провал Habr. – 2023. [Электронный ресурс] – Режим доступа: https://habr.com/ru/companies/docdoc/articles/752976/

3. Standish Group International. CHAOS 2020: Beyond Infinity [Industry report]. The Standish Group International, Inc. – 2020.

4. Fernández-Diego M. et al. An update on effort estimation in agile software development: A systematic literature review // IEEE Access. – 2020. – Т. 8. – С. 166768-166800.

5. Iqbal M. et al. Exploring issues of story-based effort estimation in Agile Software Development (ASD) //Science of Computer Programming. – 2024. – Т. 236. – С. 103114.

6. Структура распределения работ (WBS) в управлении проектами. Atlassian. – 2023. [Электронный ресурс] – Режим доступа: https://www.atlassian.com/ru/work-management/project-management/work-breakdown-structure

7. International Institute of Business Analysis (IIBA). A Guide to the Business Analysis Body of Knowledge (BABOK Guide), Version 3. Toronto: IIBA. – 2015.

8. Yang Z. et al. A survey on modern code review: Progresses, challenges and opportunities //arXiv preprint arXiv:2405.18216. – 2024.

9. Cao L. Estimating efforts for various activities in agile software development: An empirical study //IEEE Access. – 2022. – Т. 10. – С. 83311-83321.

Для заметок

Методическое издание

Дронина Анастасия Юрьевна

Методика декомпозиции и технического анализа задач в IT-проектах

Методическое пособие

Подписано в печать 24.01.2025. Гарнитура Times New Roman, Cambria.
Формат 60×84/16. Усл. п. л. 3,02. Тираж 500 экз. Заказ № 42/1.
Оригинал-макет подготовлен и тиражирован в ООО «ЭПИЦЕНТР»
308010, г. Белгород, пр-т Б. Хмельницкого, 135, офис 40
ООО «АПНИ», 308023, г. Белгород, пр-кт Богдана Хмельницкого, 135