Несколько тезисов из которых исходит автор (порядок и содержание важны):

1. Никто не знает как нужно делать правильно, все что тут написано не является истиной в последней инстанции, это субъективное неправильное мнение одного человека , можно только принимать решения и измерять их результативность, так, методом перебора, будет найден оптимальный вариант, в каждом конкретном случае.
2. Написание кода базируется на следующих принципах объектно-ориентированного программирования и паттернах проектирования:
* Абстракция - любая задача должна быть решена максимально абстрактным/простым обьектом (обьектом с минимальным количеством полей и методов), но при этом достаточным, что бы решить задачу с заданной точностью.
* KISS(keep it simple stupid) см выше
* Инкапсуляция - наличие у обьекта публичного контракта , при этом вся реализация возможностей этого обьекта скрыта, именно методы публичного контракта покрываются unit тестами, для того что бы показать как с этим обьектом нужно работать(готовые кейсы)
* Наследование - механизм выделения подмножеств объектов обладающих большей узконаправленной функциональностью , но при этом сохраняющим возможность быть использованным вместо родителя(upcast)
* Полиморфизм - реализация в зависимости от типа объекта
* Реализация - точность решения задачи(нерационально доставку пиццы осуществлять самолетом , или же доставлять нефть в Африку канистрами) , как правило тут и определяется себестоимость решения задачи.
* DRY(don’t repeat youself) наличие одного источника правды
* YAGNI Зачем?
* GRASP(ХОТЯ БЫ low coupling , high cohesion, Protected Variations)

3. Правило Парето: 20% усилий дают 80% результата (правило Парето от правила Парето : 4% усилий дают 64% результата) с точки зрения коммерческой деятельности это очень выгодный финансовый результат.
4. В одну единицу времени , на любом этапе решения задачи , у задачи должен быть один исполнитель(любой созвон по задаче двух программистов или двух аналитиков свидетельствует о том , что для задачи неверно подобран исполнитель , это контр продуктивно , нет смысла делать одну Мидл задачу Джууном и сеньором , инициатор созвона

перекладывает ответственность на более опытного коллегу, забирая его время, более опытный коллега, все равно по итогу укажет на существующее решение в коде , гораздо продуктивнее потратить время более опытного программиста на внедрение механизмов на уменьшение времени навигации по коду)

5. Технический долг - объем работы(потраченного времени), который необходимо выполнить для того что бы приступить к решению задачи или поддерживать это решение! (костыли, неочевидные реализации , экстравагантные и некие особые видения решений отличные от общепринятых в мировом профессиональном сообществе, unit тесты по 2.5 минуты)

6. Отвергаешь - предлагай!

7. Заработок это разница между доходами и расходами(влиять можно и НУЖНО на обе составляющие)

8. Unit тесты используются не для поиска ошибок , а 1. Являются частью документации по использованию компонентов(объектов или сервисов) 2. Обеспечивают гарантии того, что новый функционал или доработки не изменили поведение существующих компонентов , или новые компоненты используют существующие так как задумали создатели последних или выявление этих несоответствий

9. Любая программа это структура данных и алгоритм их обработки

10. Зарплата программиста (и аналитика) - это часть бюджета продаж.

Необходимо к внедрению в первую очередь:

1. Система оценки результативности команды за промежуток времени (Data-driven подход)

Цель : измерение количественных показателей команды за спринт для оценки принятых решений.

(Более простое определение, в правильном ли направлении движется команда, есть ли положительный эффект от принятого решения?)

Способ решения: создание математической модели, оцифровка технического процесса разработки, перевод результатов из jira в цифры, в разрезе команды и одного специалиста

Пример модели(один из , выбирать нужно лучший из предложенных): разделение задач на категории:

- junior (решения задач не создают точек изменения, простое использование компонента по аналогии, задачи вида : сделай как тут) коффициент сложности - 1
- middle (задача создает точки изменения на уровне обьекта(изменение / добавление поведения) коэффициент сложности - 2
- senior(точки изменения на уровне сервиса/логики добавление/ изменений) коэффициент сложности - 3
- lead (точки изменения на уровне архитектуры/ взаимодействия между сервисами) коэффициент сложности - 4

Реализация любой задачи предполагает несколько этапов

1. Преодоление технического долга (0 - если его нет , 1 - если реализация не самая простая и очевидная для данного уровня точности(см. Абстракция) и специалисту необходимо разобраться как это работает и почему именно так)
2. Создание структуры данных , которая необходима для решения задачи(0- структура есть , 1 - если нужно создать)
3. Реализация алгоритма обработки данных (1)
4. Создание unit - тестов для документирования и гарантий незменности достижения результата (1)

Итоговая формула для одной задачи может выглядеть так :

Points = (технический долг + структура данных + реализация алгоритма+ unit тест)*коэффициент сложности

Результат: команда за временной отрезок выполняет больший/меньший объем работы при прочих равных по сравнение с предидущим спринтом(результат - число, которое исключает множественную интерпретацию, убеждать кого то в чем то очень трудозатратно, фактами оперировать гораздо проще)

Суть модели не в правильном/инновационном/супер точном вычислении , а в ОДИНАКОВОМ вычислении / оценке любой задачи (не зависимо от сложности) от спринта к спринту.

Бонусом данная модель может использоваться для подбора исполнителей , а так же для определения уровня компетенции/квалификации специалиста по его результатам за период(KPI?)

Суть работы любого специалиста в команде при решении любых задач - это внедрение механизма , что бы последующие задачи подобного рода (из классификации) были переведены в более низкий класс, и соответсвенно могли быть выполнены менее квалифицированным специалистом.

2. Система определения бизнес-ценности задачи

Профессиональный аналитик может сэкономить команде несколько сеньор разработчиков, обратное утверждение так же справедливо.

* Четкое понимание того, в чем заключается бизнес ценность задачи для заказчика , и именно это должно быть отражено в постановке.
* Четкое понимание того, что аналитик , поставивший задачу, в большей степени определяет время программиста , решающего задачу, независимо от уровня квалификации обоих , практика показывает ,что данное взаимодействие не всегда конструктивно , как минимум нужно отслеживать эффективность и проблематику этих взаимодействий!!!
* Классификация задач применима из предидущих пунктов , эффективнее постановку для senior задач писать аналитику с senior-ным опытом в проекте.
* Однозначность постановки (при постановке задач программисту необходимо понимать, какую проблему необходимо решить, что является/может являться результатом выполнения задачи, если задача относится к визуалу хотелось бы понимать требования к внешнему виду), то есть постановка должна быть одинаково интерпретирована и программистом и тестировщиком.
* Время разработки может существенно сократить постановки вида : сделай как тут.

3. Внедрение механизма решения проблем

Цель : Проведение анализа результатов спринта в разрезе команды и каждого специалиста , должен быть внедрен механизм оценки действий того или иного специалиста приведших к плохим результатам , необходимо понимать что приводит к уменьшению производительности, для устранения первопричины.

Реализация: Ретроспектива после каждого спринта с целью систематизации и классификации проблем. Пересечение озвученных проблем разумно рассматривать в приоритетном порядке.

Дополнительно после общего обсуждения есть смысл более детально разбираться с низом списка и выяснять причины более комплексно(объективные и субъективные)

Внедрение во «вторую» очередь (на самом деле в нулевую, это базис)

4. «Соглашение о коде»

Механизм объединения опыта и лучших практик участников команды , по решению стандартных задач/проблем одними и теми же способами. Сюда могут относиться такие вопросы как (тут описаны как и общие вопросы, так и проблемные моменты с которыми столкнулся я):

* Принципы ООП и GRASP (см тезис 2)
* Расположение объектов (Котлин не следит за наименование пакетов и реальными путями , при копировании может складываться впечатление что пакеты common могут зависит от web, неочевидные решения по расположению объектов , монолитная или микросервисная модель , по главному объекту в иерархии , по модулю, и тд)
* Именование объектов в зависимости от назначения (5 классов с названием security с частичным дублированием(вроде и KMP вроде и нет), классы *Util могут иметь такой же набор импортов как и класс *Service , менять состояние объекта в БД то есть являться классом по работе с бизнес логикой, хотя классы Util не должны зависеть от экземпляра, паттерны GoF для экономии времени,…)
* Количество зависимостей в классе(следить за импортами , что бы не допускать большой связности и низкого зацепления)

* Сигнатуры методов (могут быть методы с 8-10 входными параметрами , это увеличивает когнитивную нагрузку и может приводить к нарушению GRASP принципов связности и зацепления , как тестировать такие методы ?, по правилам комбинаторики что бы покрыти полностью тестами такой метод нужно очень много кода , необходимо пытаться уменьшать связность (путем внедрения GRASP посредник или чистая выдумка)
* Глубина стека вызовов (иногда для результата нужно вызвать большое количество методов, как покрывать такой код тестами , пытаться следить за уровне вложенности вызовов и уменьшать его, чем меньше действий тем больше вероятность успешного выполнения)
* Количество операторов ветвления
* … список неполный, главный посыл : это не создание правил которые все дружно будут нарушать , а некая попытка объединить опыт , то есть делать единообразно максимально простым и эффективным способом

На мой взгляд наличие некоего соглашения о том как команда решает типовые проблемы дало бы ряд преимуществ в скорости разработки:
* Более быстрая навигация по коду(что где расположено, по названию понимать за что отвечает тот или иной класс, …)
* Более быстрое преодоление технического долга
* Соблюдение DRY , не будут создаваться дублирующие решения
* Упрощение процедуры ревью (соглашение о коде не нарушено: убедиться в принципе абстрактности , убедиться что задача решена не дублирующимся способом , …)

5. Организационные и технические решения , которые могли бы положительно повлиять на продуктивность

* Есть преимущества в том , чтобы каждый этап задачи(постановка, реализация, тестирование и др) реализовывать специалистом с соответствующим уровнем компетенции, уровень компетенции определяется результатами за предыдущие спринты. Пример: уточнением данных, не требующих специфических знаний или навыков должен заниматься специалист с наименьшей зп, ничего личного только бизнес. Ситуация когда разработчик вместе с тестировщиком созваниваются и решают , а что же имелось ввиду в постановке и

как мы будем это все разрабатывать и тестировать , как минимум очень дорого , этим должен заниматься сотрудник с наименьшей квалификацией(зп)

* Атомарность задачи. Есть преимущества в том что бы декомпозировать бизнес задачу на более атомарные логические подзадачи , это не изменяет обьем работы у программиста, но позволит :

* Оценить задачу/задачи более точно (время, сложность ..)

* Ревью проще и быстрее

* Передавать задачи в тестирование раньше и их тестирование будет проще по обьему и меньше по времени

* Уменьшить время между мерджами , что поспособствует возникновению меньшего числа конфликтов ввиду того, что на подзадачу тратиться меньше времени и пул-реквест оформляется быстрее , кроме того коммит по составу файлов меньше

* В целом уменьшение когнитивной нагрузки при разработке , ревью и тестировании

* Пример: импорт/экспорт или экспорт в форматы xml, json, csv, xls с точки зрения бизнеса задачи едины , но экспорт может делаться несколько часов , а импорт несколько дней, или же доработка в xml займёт несколько часов , а в формате json , например , несколько дней, таким образом , условно , к моменту реализации импорта , экспорт уже может быть протестирован и переведён в релиз , или к моменту реализации json , xml будет протестирован и переведён в релиз , что может снизить нагрузку на тестирование в пиках , когда проходят массовые апрувы по ревью перед релизами (работы на выходных), понятное дело подход не панацения, но условия в целом будут более благоприятными для нужного результата

* Тестирование. Использование unit тестирования , тоесть писать тест на участок кода который разрабатывается , а создание тестового окружения для запуска должно быть передано фреймворку , вместо этого , во всех тестах связанных с DataMaps у нас используется интеграционное тестирование (совокупность нескольких сервисов), тоесть создание тестового окружения выполняется руками программиста, и поэтому запуск теста соизмерим по времени со стартом сервера для того что бы поднять практически реальный контекст di контейнера с большим количеством зависимостей, любой тест с datamaps работает около 2.5 минут. Например тест с импортом/экспортом в csv занимает менее 10 секунд ввиду малой связности компонента CsvUtil. Тестирование datamap должны быть доработано механизмами создания тестового контекста средствами фреймворка @sql(query.sql) может сразу создавать необходимое состояние бд с необходимыми данными для тестирования именно того функционала который находится в разработке. Совершенно

точно ответственным за разработку в команде более опытным специалистам нужно взять за основу любой труд по тестированию(самый масштабный на данный момент - Test Driven Development, не обязательно его , но главное правильное использование) и внедрить в процесс тестирования обеспечив ключевые принципы и механизмы этого подхода.