Хорошее Техническое Задание

Проблематика

Идеальная картина выглядит так: есть *Заказчик*, *Аналитик* и *Программист*. *Заказчик* рассказывает *Аналитику*, **что** он хочет получить, *Аналитик* придумывает, **как** это сделать, наконец, *Программист* кодирует то, что написал ему аналитик.
Проблемы идеальной картины:

1. Неадекватный *Заказчик*. Известнейший пример неадекватного внутреннего *Заказчика* — Джон Скалли на посту CEO Apple. Речевой интерфейс за 30 лет до того, как он реально мог быть сделан: http://appleinsider.ru/video/mechtat-polezno-apple-predskazala-siri-v-1987-godu.html
2. X-Y проблема. *Заказчик* хочет решить проблему X, и предполагает, что для этого хорошо подходит решение Y. Для специалистов X и Y никак не связаны, и решение X через Y выглядит странным и неестественным, но в ТЗ написано именно: нужно, чтобы программа умела делать Y.
3. Размытая граница между *Аналитиком* и *Программистом*. Для примера возьмём ТЗ в стиле RUP. Сначала требования *Заказчика* описываются как прецеденты (use cases), затем формируется техническое решение, которое иллюстрируется UML-диаграммами. Затем *Программист* реализует это решение. Кто занимается проектированием, кто рисует UML-диаграммы?
4. Испорченный телефон: *Аналитик* может искажать информацию. Он может неправильно понять *Заказчика*, он может домыслить за *Заказчика*. Открытый вопрос: можно ли воспитать правильного *Аналитика*? Есть ли критерии правильного *Аналитика*? Можно ли избавиться от магии в этом вопросе?
5. Абстрактный *Аналитик*. *Аналитик* слишком оторван от реалий разработки, поэтому предлагает сложные решения проблем *Заказчика*.

Гибкий подход

Решение, предложенное в гибких подходах, кардинально донельзя. Разработка итеративна, и Заказчик всегда доступен для команды. Заказчик рассказывает своё видение с помощью пользовательских историй, затем *Программист* уточняет и детализирует эти истории. В гибких методологиях речь идёт о кросс-функциональной команде, поэтому опытные специалисты могут выполнять функции *Аналитиков* и *Программистов*.
Проблемы гибкого подхода:

1. Размер проектов. Статистика противоречива, но в общем и целом известна проблема масштабируемости гибких методологий. Для больших проектов это может не заработать.
   «ЗА»: https://www.infoq.com/articles/standish-chaos-2015
   «ПРОТИВ»: https://www.versionone.com/assets/img/files/CHAOSManifesto2013.pdf

2. Внутренние требований компаний на документацию. Гибкая документация не строга, поскольку работающий код важнее, чем полноценная документация. Но требования компании могут требовать строгой документации и строгого ТЗ.
3. Стоимость разработки. Долгосрочное планирование не приветствуется, но иногда точно и заранее известно, что потребуется. Можно запланировать и реализовать заранее.
4. Плохие пользовательские истории. Надо учиться, и это непростой процесс.

Решения?