http://blog.grine.net/2016/04/27/waterfall/, апрель 2016

Управление разработкой больших компьютерных систем

Dr. Winston W. Royce

Опубликовано в Proceedings, IEEE WESCON, August 1970, pages 1-9

Перевод: Григорий Грин, редактор перевода: Нэлли Грин

Введение

Я собираюсь описать свой взгляд на управление разработкой больших программных систем. В течение последних девяти лет я был вовлечён в ряд проектов, связанных с планированием миссий космических кораблей, управления ими и послеполётным анализом. В этих проектах я видел разный уровень успеха относительно запуска в эксплуатацию вовремя и в рамках бюджета. В результате этого опыта у меня появились некоторые соображения, которыми я здесь хочу поделиться.

Функции в разработке компьютерных программ

В любой компьютерной разработке, вне зависимости от размера и сложности, можно выделить два шага. Первый – это **анализ**, за которым следует **кодирование**, как представлено на рис. 1. Столь простая концепция реализации - это вообщем-то всё что нужно, если задача достаточно мала, и если готовый продукт будет использоваться теми, кто его строил. А также, это те два вида деятельности, за которые заказчики готовы платить, так как оба шага включают в себя творческую работу, которая вносит вклад в полезность конечного продукта. Однако, если план реализации большого проекта включает в себя лишь эти шаги, то проект обречён на провал. Необходимы многие дополнительные шаги, ни один из которых не влияет столь непосредственно на конечный продукт как анализ и кодирование, но при этом существенно увеличивают расходы. Типичный заказчик предпочтёт не оплачивать эти шаги, а типичный исполнитель – не выполнять их. Главная функция менеджмента состоит в том, чтоб продать необходимость дополнительных работ обеим группам, а затем обеспечить их выполнение на стороне исполнителя.

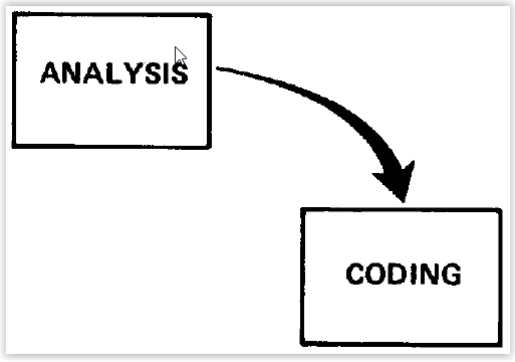

Рисунок 1 Шаги реализации для небольшой программы, предназначенной для внутреннего использования

Более претенциозный подход к разработке представлен на рис. 2. Шаги анализа и кодирования по-прежнему на месте, но им предшествуют два уровня **анализа требований**,

между ними вклинивается шаг **дизайна**, а завершает всё шаг **тестирования**. Эти дополнения рассматриваются отдельно от анализа и кодирования, поскольку они иные по способу их выполнения. Их надо планировать иначе и включать в них других специалистов, чтобы использовать ресурсы проекта более рационально.

На рис. 3 изображены итеративные связи между последовательными шагами на этой схеме. Последовательность шагов основана на следующей идее: завершение одного шага может повлиять на решения, принятые в предыдущем шаге и, естественно, на следующий шаг, но редко на более удалённые шаги. Достоинство этого в том, что по мере того как дизайн конкретизируется, изменения уже принятых решений сводятся до контролируемого минимума. В любой момент времени после завершения анализа требований существует ясная базовая линия (base line), к которой можно вернуться в случае непредвиденных трудностей с дизайном. Это эффективная запасная позиция, которая максимизирует объём работы, сделанной на ранних стадиях проекта, результаты которой можно сохранить и использовать.

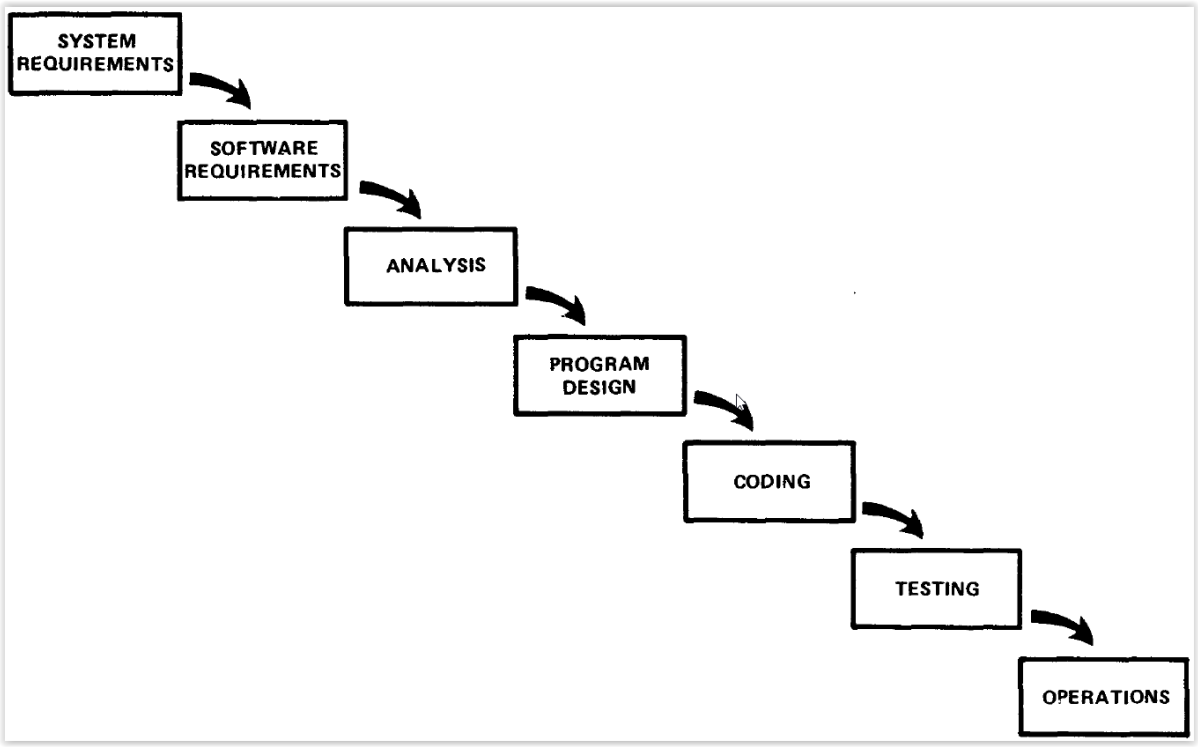

Рисунок 2 Шаги реализации для большой системы, поставляемой заказчику

Я верю в описанную концепцию, но реализация, представленная выше рискованна и чревата провалом. Проблема показана на рис. 4. Фаза тестирования, которая расположена в самом конце цикла разработки, это первый момент в проекте, когда становятся видимы отклонения по таким параметрам как скорость работы, объём хранения, ввод/вывод и т.д. от первоначальных результатов анализа. Эти показатели невозможно точно проанализировать. Их нельзя получить, например, как результаты решения дифференциальных уравнений в математической физике. Однако, если эти параметры выйдут за пределы некоторых внешних ограничений, придётся переделывать весь дизайн. Простой патч или изолированное улучшение в одном из модулей не решит такие проблемы. Изменения в дизайне будут, скорее всего, значительны, так как те требования, на которых основывался дизайн и которые являлись стержнем всего, оказались нарушены. Или придётся модифицировать требования, или

существенно переделывать дизайн. В результате, проект вернётся в исходную точку, и можно ожидать превышения бюджета и сроков на 100%.

Читатель может заметить, что мы обошли шаги анализа и кодирования. Естественно, невозможно построить продукт без этих шагов, но, однако управлять этими шагами достаточно легко, и они мало влияют на требования, дизайн и тестирование. У меня есть опыт, когда целые отделы были заняты вычислением орбит, определением положения космического корабля, математической оптимизацией полезного груза и т.д., но, когда эти отделы заканчивали свою важную и трудную работу, результирующий программный код сводился к нескольким строкам арифметических операций. Если в процессе сложной аналитической работы была допущена ошибка, её коррекция будет изменением нескольких строк программного кода, без разрушительных последствий в других местах.

Однако, я верю, что представленный подход фундаментально разумен. Остальная часть этой статьи представляет пять дополнительных модификаций, которые необходимо ввести в концепцию, чтобы устранить большинство рисков при разработке.

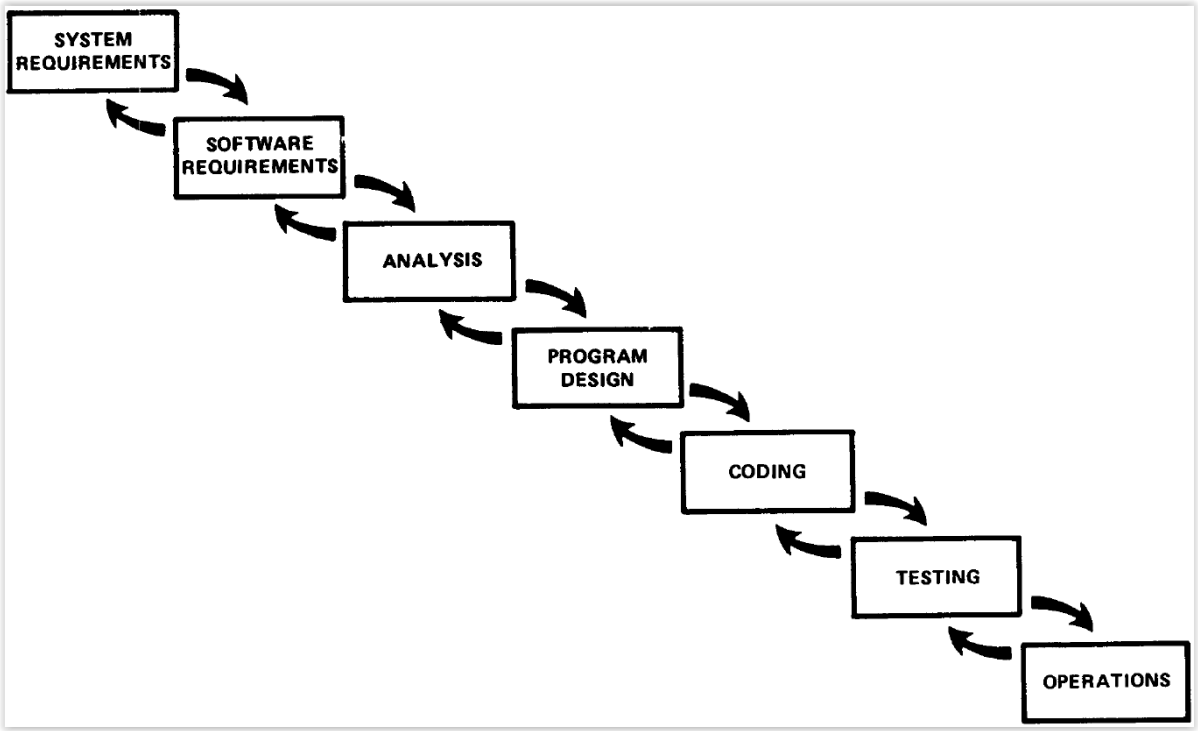

Рисунок 3 Хочется надеяться, что итеративное взаимодействие между фазами ограничено только непосредственно граничащими шагами

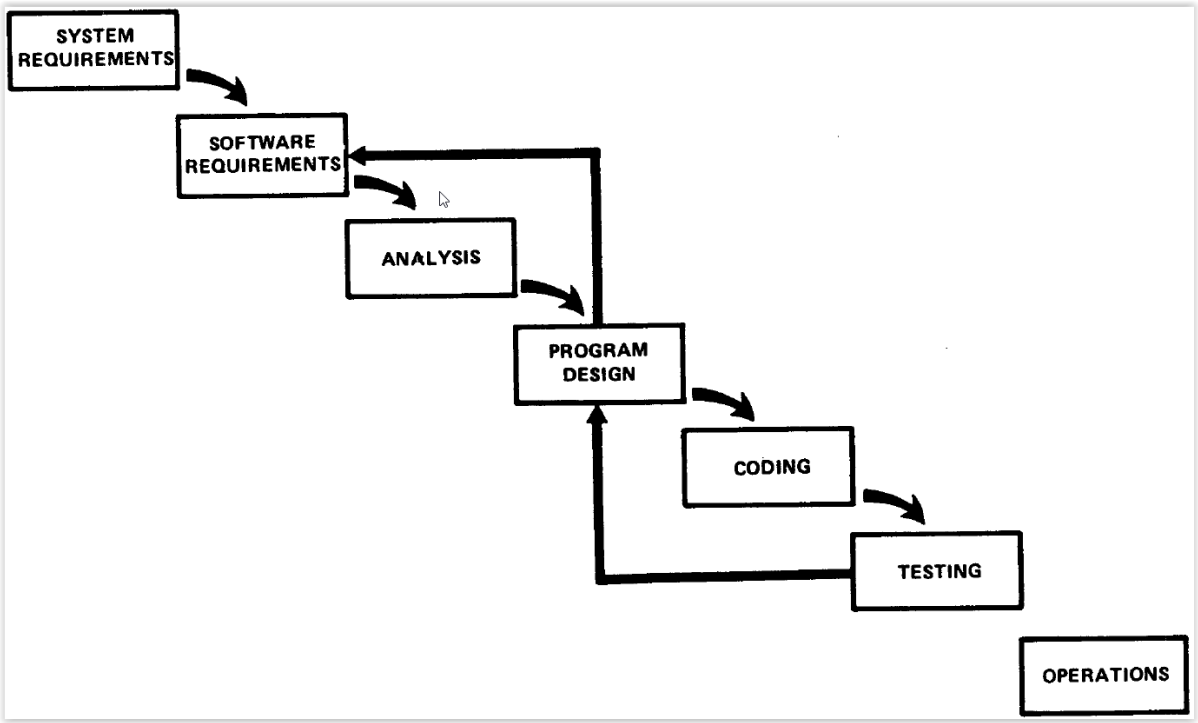

Рисунок 4 К сожалению, в примере на иллюстрации, изменения, вызванные дизайном, не ограничиваются лишь соседними шагами

Шаг 1: Начинайте с дизайна

Первый шаг по улучшению процесса представлен на рис. 5. Между требованиями и анализом добавлен шаг разработки предварительного дизайна. Этот подход можно критиковать на базе того, что разработчик предварительного дизайна находится в относительном вакууме требований без необходимого анализа. В результате предварительный дизайн может оказаться ошибочным, если сравнивать его с конечным дизайном, выполненным после фазы анализа. Критика верна, но она не учитывает важного обстоятельства. Пользуясь этим методом, разработчик обеспечивает, чтобы программа не провалилась из-за скорости её работы, объёма хранимых данных или слишком большого потока данных. При переходе к фазе анализа, разработчик должен передать аналитику информацию об ограничениях относительно производительности, хранения данных и другие эксплуатационные ограничения таким образом, чтобы тот смог сделать соответствующие выводы. Если аналитику действительно нужно больше таких ресурсов чтобы решить свои уравнения, они будут одновременно отобраны у других аналитиков. Таким образом, все аналитики и все разработчики участвуют в разумном процессе дизайна, кульминацией которого является правильное распределение вычислительных ресурсов и памяти. Если суммарных ресурсов не хватает, или же первоначальный дизайн-эмбрион неверен, это будет замечено рано, и исправлено путём коррекции требований и поправок в дизайне ещё до того, как будут начаты настоящий дизайн, кодирование и тестирование.

Как реализовывается эта процедура? Необходимы следующие шаги:

1) **Начинайте процесс дизайна силами дизайнеров приложений**, а не аналитиков или программистов.

2) **Принимайте решения по дизайну** даже рискуя совершить ошибку. Это включает определение входов и выходов, интерфейсов с операционной системой, базы данных, распределение вычислительного времени, функции и т.д.
3) **Составьте обзорный документ**, который понятен, информативен и актуален. Каждый сотрудник в проекте должен иметь элементарные представления обо всей системе в целом. По крайней мере один человек должен иметь глубокое представление о системе, которое частично приходит от написания обзорного документа.

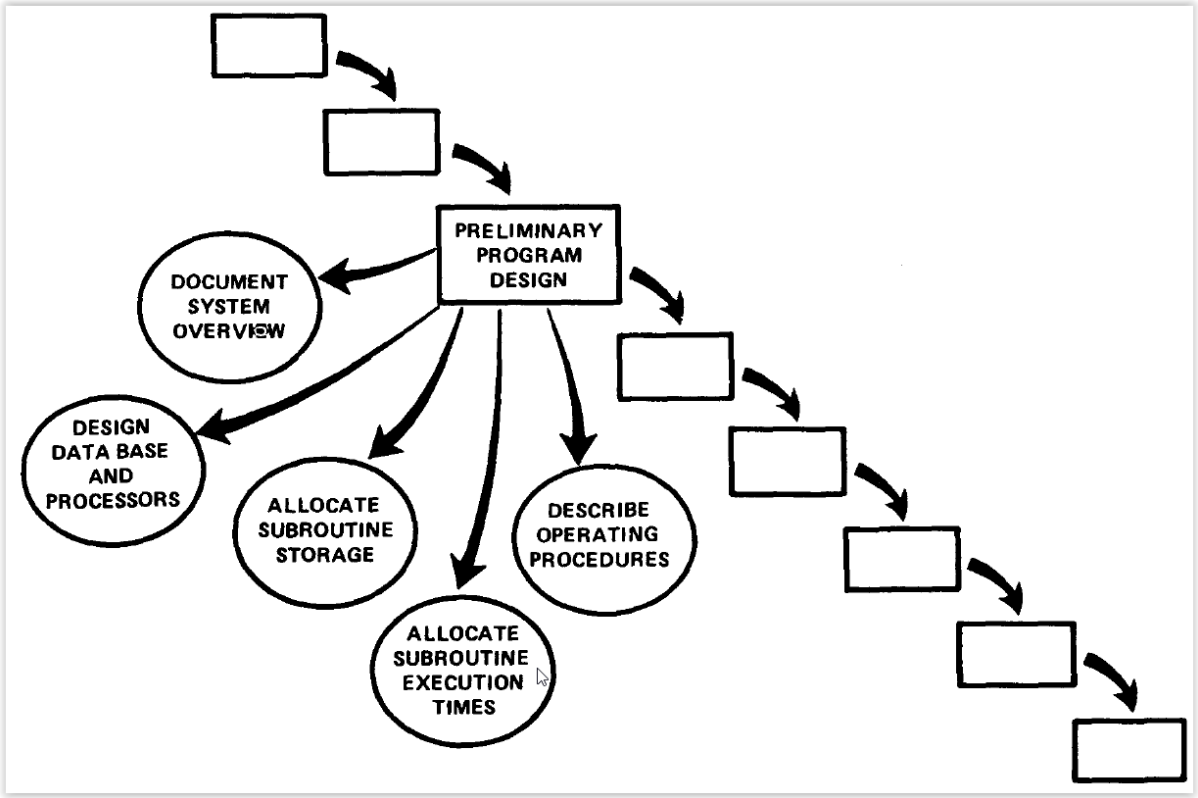

Рисунок 5 Шаг 1: Проведите предварительный дизайн до начала анализа

Шаг 2: Документируйте дизайн

Тут логично задаться вопросом – «сколько документации?». Мой ответ: «довольно много»; во всяком случае больше, чем большинство программистов, дизайнеров и аналитиков написали бы, если бы были предоставлены сами себе. Первое правило управления проектами разработки программ – это бескомпромиссное выполнение требований по документированию.

Иногда меня приглашают проанализировать состояние других проектов. Мой первый шаг – это проверить состояние документации. Если документация в плачевном состоянии, моя рекомендация проста. Заменить руководство проектом. Остановить всю работу, не связанную с документированием. Привести документацию в приемлемое состояние. Управление программным обеспечением просто невозможно без высокой степени документирования. В качестве примера, позвольте мне предложить одну оценку для сравнения. Для закупки компьютерного оборудования на 5 миллионов долларов, я бы ожидал, что спецификация на 30 страницах предоставит достаточно деталей для управления процессом закупки. Чтоб сделать то же с программным обеспечением на 5 миллионов долларов, я бы оценил

необходимый объем документации в 1500 страниц для достижения сравнимой степени контроля.

Почему так много документации?

1) Каждый дизайнер должен взаимодействовать с соседними (по интерфейсам) дизайнерами, со своим начальником и, возможно, с заказчиком. Устная коммуникация слишком нематериальна, чтобы обеспечить определения интерфейсов или решения по управлению. Описание на бумаге вынуждает дизайнера занять однозначную позицию и предоставить осязаемые доказательства завершения работы. Это не даёт дизайнеру прятаться за высказыванием «работа на 90% готова» в течение нескольких месяцев.
2) На ранних фазах разработки документация является и спецификацией, и дизайном. До того, как начнётся кодирование эти три слова (документация, спецификация, дизайн) означают одно и то же. Если плоха документация, плох и дизайн. Если документации нет, дизайна тоже нет. Если только люди, думающие и говорящие о дизайне, что уже неплохо, но недостаточно.
3) Документация начинает приносить реальную пользу, исчисляемую в деньгах, на дальнейших фазах процесса – во время тестирования, эксплуатации и ре-дизайна. Значимость документации может быть показана в трёх конкретных осязаемых ситуациях, с которыми сталкивается каждый менеджер проектов.
    a. **Во время фазы тестирования**, с хорошей документацией менеджер может сфокусировать персонал на ошибках в программе. Без хорошей документации каждая ошибка, большая или маленькая, анализируется одним человеком – тем, кто, вероятно, и сделал эту ошибку, потому что он единственный, кто понимает эту область в программе.
    b. **Во время эксплуатации**, с хорошей документацией менеджер может использовать персонал, ориентированный на эксплуатацию чтобы сопровождать программу, при этом делая работу лучше и дешевле. Без хорошей документации, программу вынуждены сопровождать те, кто её разработали. Обычно, эти люди не заинтересованы в сопровождении, и не выполняют работу столь эффективно, как те, кто сфокусирован на сопровождении. Нужно сказать в этой связи, что во время эксплуатации, чтобы не случилось, сначала обвиняется программа. Чтоб или оправдать программу, или подтвердить претензию документация должна «говорить» чётко и ясно.
    c. **После первоначального периода эксплуатации**, когда пора вносить изменения в программу, хорошая документация позволяет эффективный редизайн, модификацию и подгонку в боевой обстановке. Если документации нет, обычно весь софт приходится выбрасывать и переписывать, даже в случае скромных изменений.

На рис. 6 изображен план документации, подогнанный к шагам, описанным выше. Заметьте, что производятся 6 документов, и в момент поставки финального продукта документы 1, 3, 4, 5 и 6 обновлены и являются актуальными.

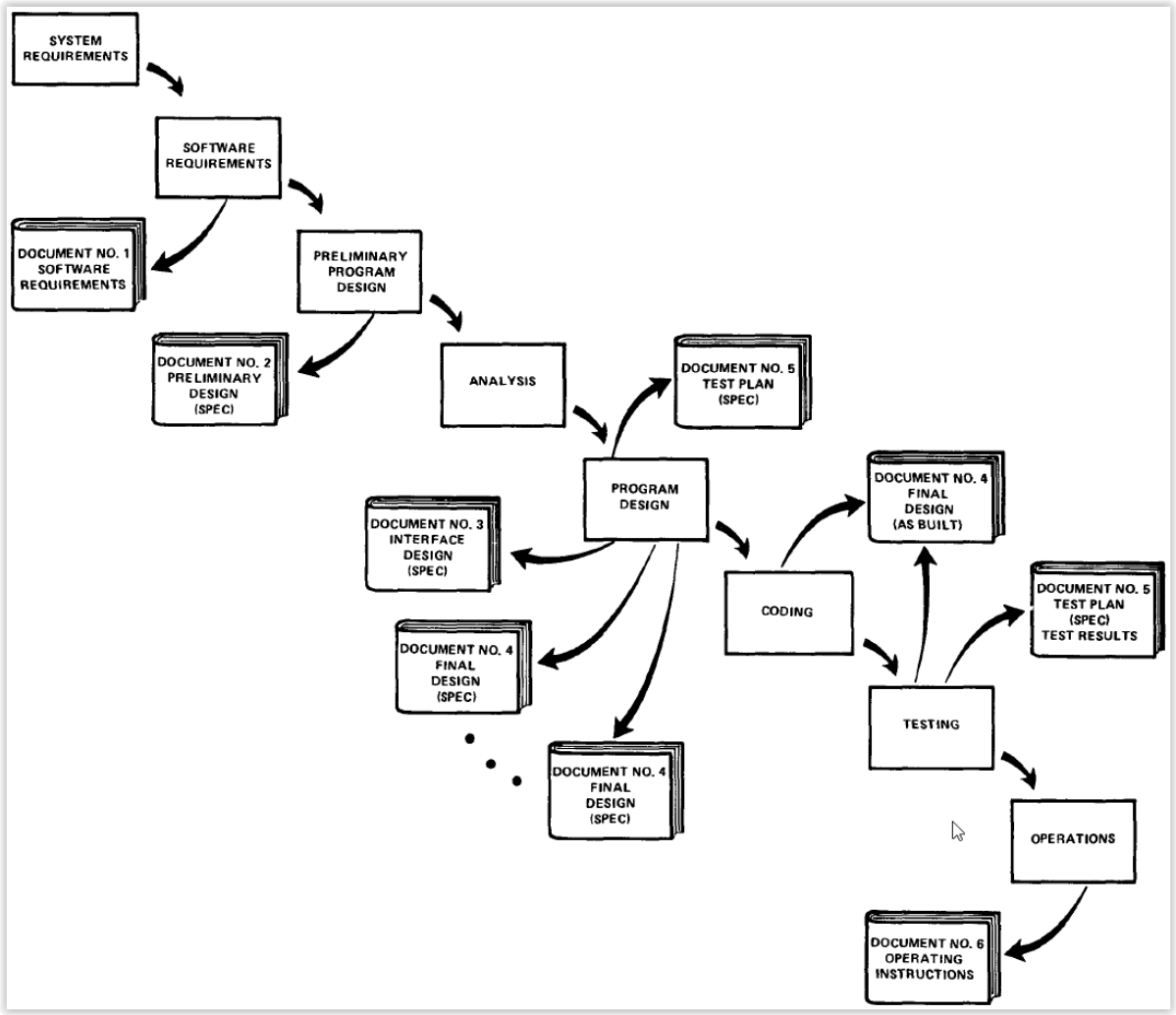

Рисунок 6 Шаг 2: Обеспечьте наличие полной и актуальной документации. Необходимы по крайней мере 6 обозначенных документов.

Шаг 3: Делайте это дважды

После документации следующий критерий успеха вращается вокруг того, является ли продукт совершенно новым. Если программа разрабывается с нуля, организуйте процесс так, чтобы клиент получил в эксплуатацию как минимум вторую версию, по крайней мере того, что касается критических областей дизайна/эксплуатации. Рис. 7 демонстрирует как это можно сделать с помощью моделирования. Заметьте, что это просто весь процесс, выполненный в миниатюре за время, являющееся небольшим относительно всего процесса. Природа этой миниатюры может быть разной в зависимости от имеющегося времени и сути критических проблем, подлежащих моделированию. Если весь проект расчитан на 30 месяцев, то для пилотной модели можно выделить 10 месяцев. Для этой длительности могут использоваться вполне формальные методы контроля, процедур документации и т.д. Если же общая длительность составляет лишь 12 месяцев, то пилот можно сжать, скажем, в 3 месяца, чтобы получить подходящий рычаг для основного проекта. В этом случае от вовлечённого персонала требуется особый вид широких компетенций. Они должны интуитивно чувствовать анализ, кодирование и дизайн. Они должны быстро чувствовать проблемные места в дизайне, моделировать их, моделировать их альтернативы, игнорировать простые аспекты дизайна, которые не стоят дополнительного изучения в этот ранний момент, и в результате получить безошибочную программу. В конечном счёте, смысл всего этого моделирования в том, чтобы

вопросы ресурсов вычисления и хранения данных, которые иначе пришлось бы решать чисто теоретически, теперь могли изучаться практически и точно. Без такого моделирования, менеджер проекта вынужден полагаться на суждения своих сотрудников. Имея фазу моделирования, он может по крайней мере провести экспериментальный тест некоторых ключевых гипотез и локализовать те места, где действительно надо полагаться на экспертное мнение. Оно, однако, в области компьютерного дизайна всегда слишком оптимистично (к примеру во вопросам оценки максимального стартового веса, общих ожидаемых расходов или победителя лошадиных скачек).

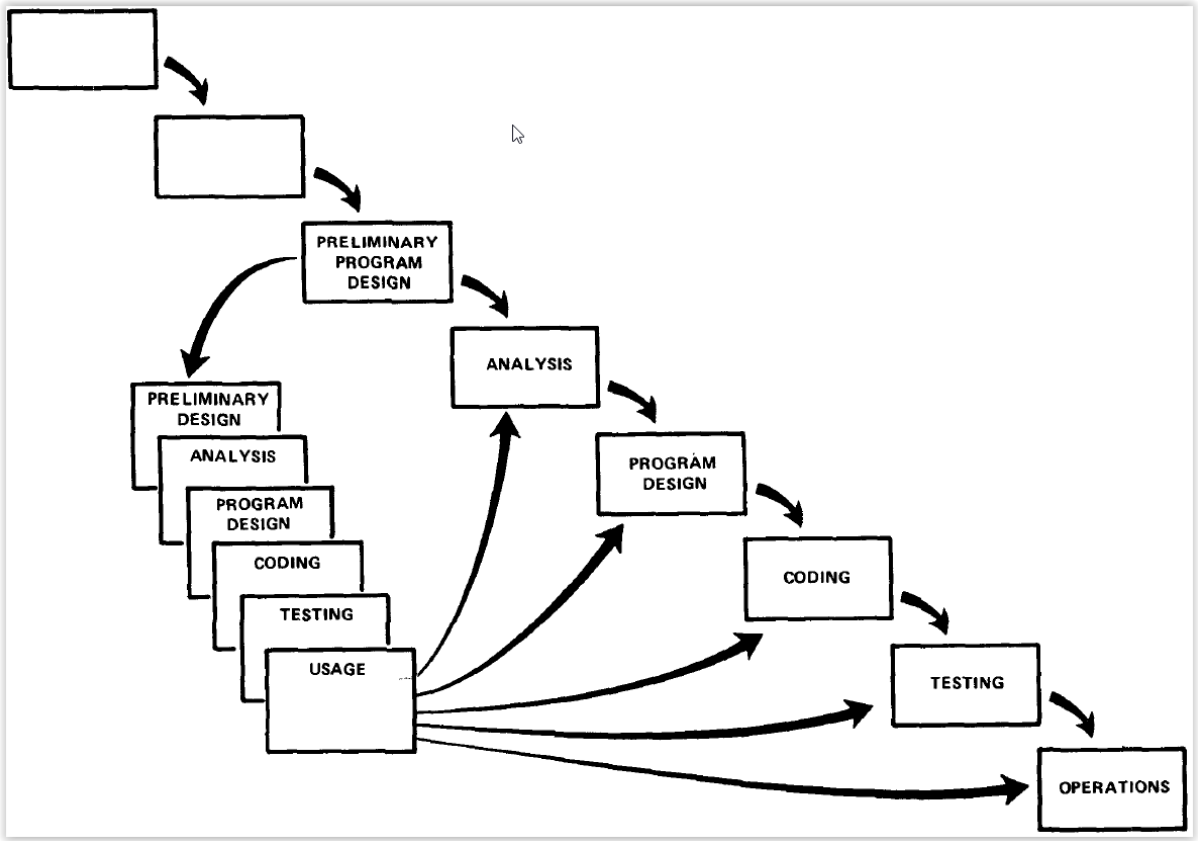

Рисунок 7 Шаг 3: Ввыполните работу дважды: результаты первого прогона играют для всего проекта роль модели

Шаг 4: Планируйте, контролируйте и отслеживайте тестирование

Без вариантов, самый большой потребитель проектных ресурсов, будь то трудозатраты, машинное время или способность менеджера принимать решения – это фаза тестирования. Это фаза наивысшего риска относительно бюджета и длительности проекта. Она происходит в конце проекта, когда возможности альтернативных решений сильно ограничены, если вообще имеются.

Предыдущие три совета (начинать дизайн до анализа и кодирования, документировать детально и строить прототип) все нацелены на нахождение и устранение проблем ещё до начала фазы тестирования. Однако, даже если всё это сделать, всё равно в конце остаётся фаза тестирования, и в ней всё равно происходят очень важные вещи. Рис. 8 перечисляет некоторые дополнительные аспекты тестирования. При планировании тестирования я предлагаю учитывать следующее.

1) Многие части процесса тестирования лучше всего поручить людям, которые не были заняты в построении первоначального дизайна. Если говорят, что только дизайнер может качественно протестировать, потому что только он знает как программа устроена, то это знак того, что документация выполнена ненадлежащим образом. Имея хорошую документацию, лучше использовать специалиста по качеству ПО, который, по моему мнению, лучше справится с тестированием, чем дизайнер.
2) Большинство ошибок тривиальны и могут быть легко обнаружены визуальной инспекцией. Всё описание дизайна и весь код должны быть визуально просканированы специалистом, который не участвовал в создании соответствующего модуля. Он легко найдёт такие вещи, как пропущенный знак минус, пропущенное умножение на 2, прыжки по неправильному адресу, и т.д. Не используйте компьютер для нахождения таких дефектов – это слишком дорого.
3) Тестируйте каждый логический путь через программу по крайней мере один раз с помощью какой-нибудь числовой проверки. Если бы я был заказчиком, я бы не принял поставку, пока такая процедура не выполнена и не подтверждена. Этот шаг выявит большинство ошибок в кодировании.

   Процедура тестирования звучит просто, но для большой и сложной программы это довольно трудно – пропахать все ветки кода с контроллируемыми входными значениями. Найдутся такие, кто будет говорить что это практически невозможно. Несмотря на это, я настаиваю на своей рекомендации, что каждый маршрут сквозь код подлежит хотя бы одной проверке.
4) После того, как простые ошибки устранены (а это большинство, скрывающее более серьёзные ошибки), программа переходит в финальную стадию тестирования. В определённый момент времени и в руках подходящего специалиста копмьютер сам по себе является лучшим инструментом проверки. Ключевое управленческое решение здесь – кто и когда выполнит финальную проверку и каковы показатели того, что тестирование завершено.

Шаг 5: Вовлекайте заказчика

По каким-то причинам то, что именно программа должна делать, остаётся предметом широкой интерпретации даже после предварительного согласования. Важно вовлечь заказчика формальным способом, с тем чтобы он подтвердил своё согласие на ранних фазах, до финальной поставки. Давать подрядчику полную свободу между формулировкой требований и вводом в эксплуатацию – это запрограммированные неприятности. Рис. 9 показывает три точки после формулировки требований, где проницательность, суждения и поддержка заказчика может поддержать проект разработки.

Итоги

Рис 10. обобщает пять шагов, которые я считаю необходимыми для преобразования рискованного проекта разработки в такой, который произведёт желаемый результат. Я подчеркну, что каждый элемент стоит определённую сумму денег. Если бы более простой процесс, без этих пяти шагов, работал успешно, тогда, конечно, эти деньги не стоило бы тратить. В моём опыте, однако, более простой метод никогда не работал для больших проектов, и затраты на переделки значительно превышали те, что понадобились бы на описанные пять шагов.

http://blog.grine.net/2016/04/27/waterfall/, апрель 2016

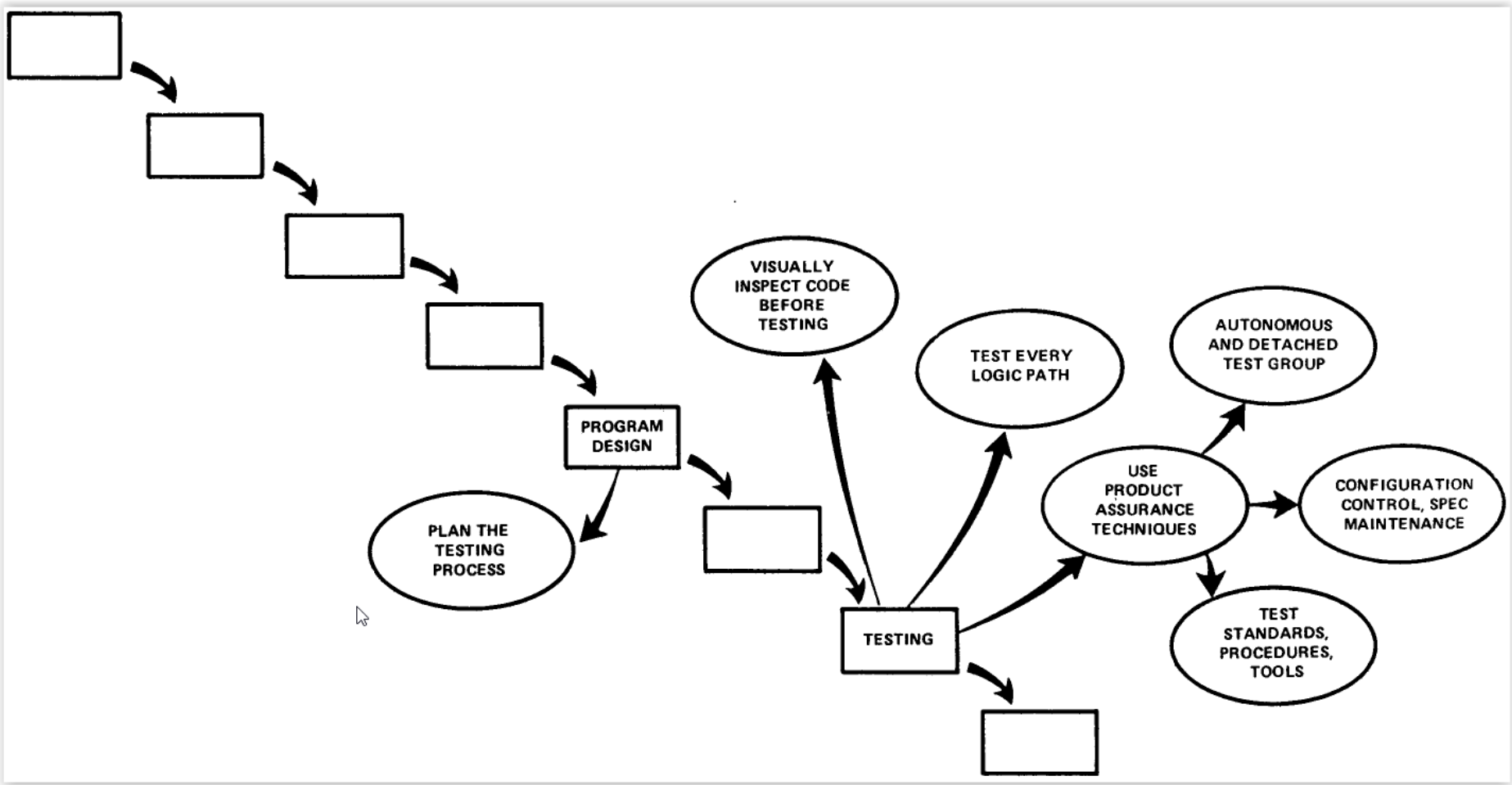

Рисунок 8 Шаг 4: Планируйте, контролируйте и отслеживайте процесс тестирования

http://blog.grine.net/2016/04/27/waterfall/, апрель 2016

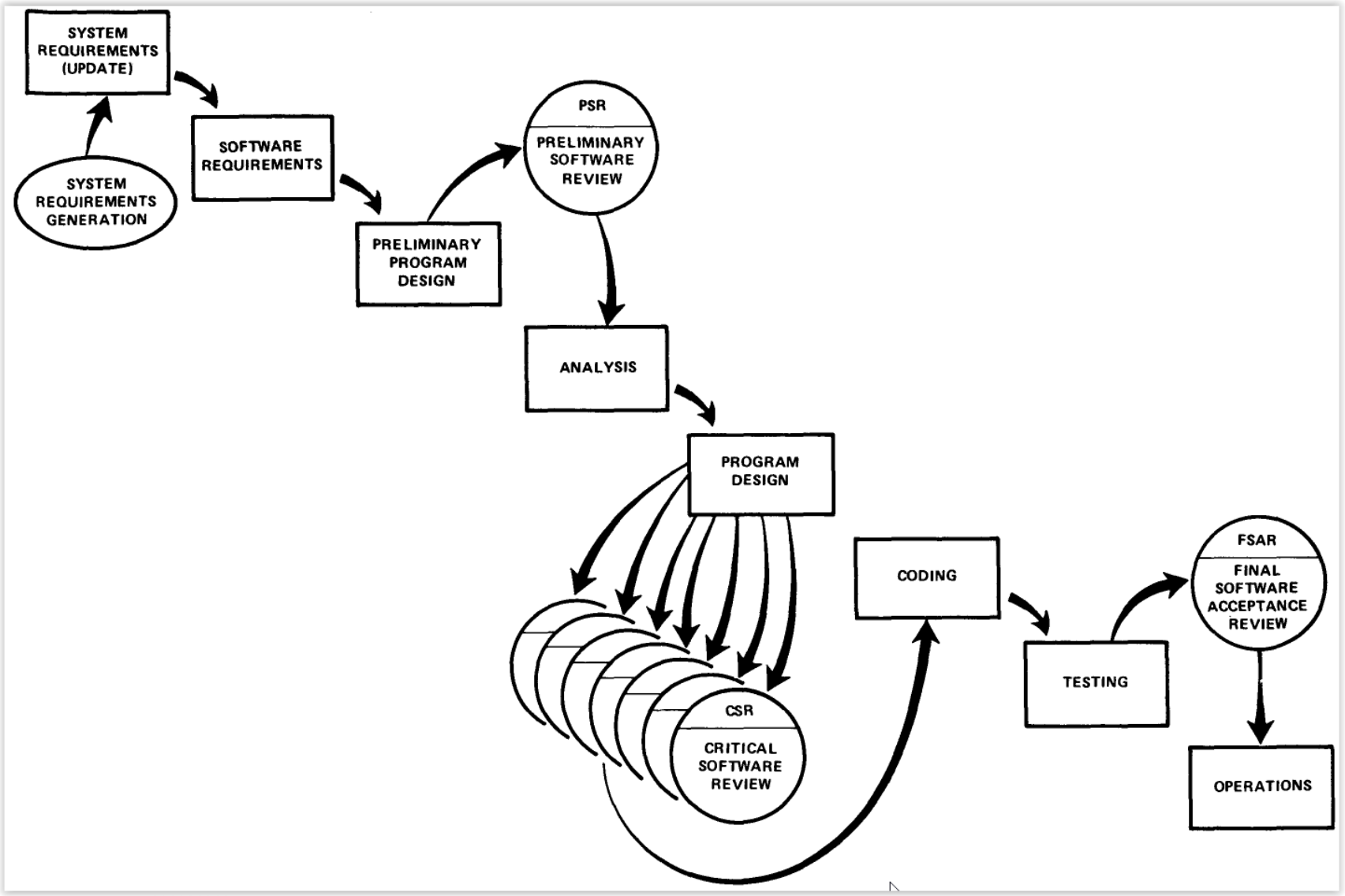

Рисунок 9 Шаг 5: Вовлекайте заказчика - участие заказчика должно быть формальным, глубоким и постоянным

http://blog.grine.net/2016/04/27/waterfall/, апрель 2016

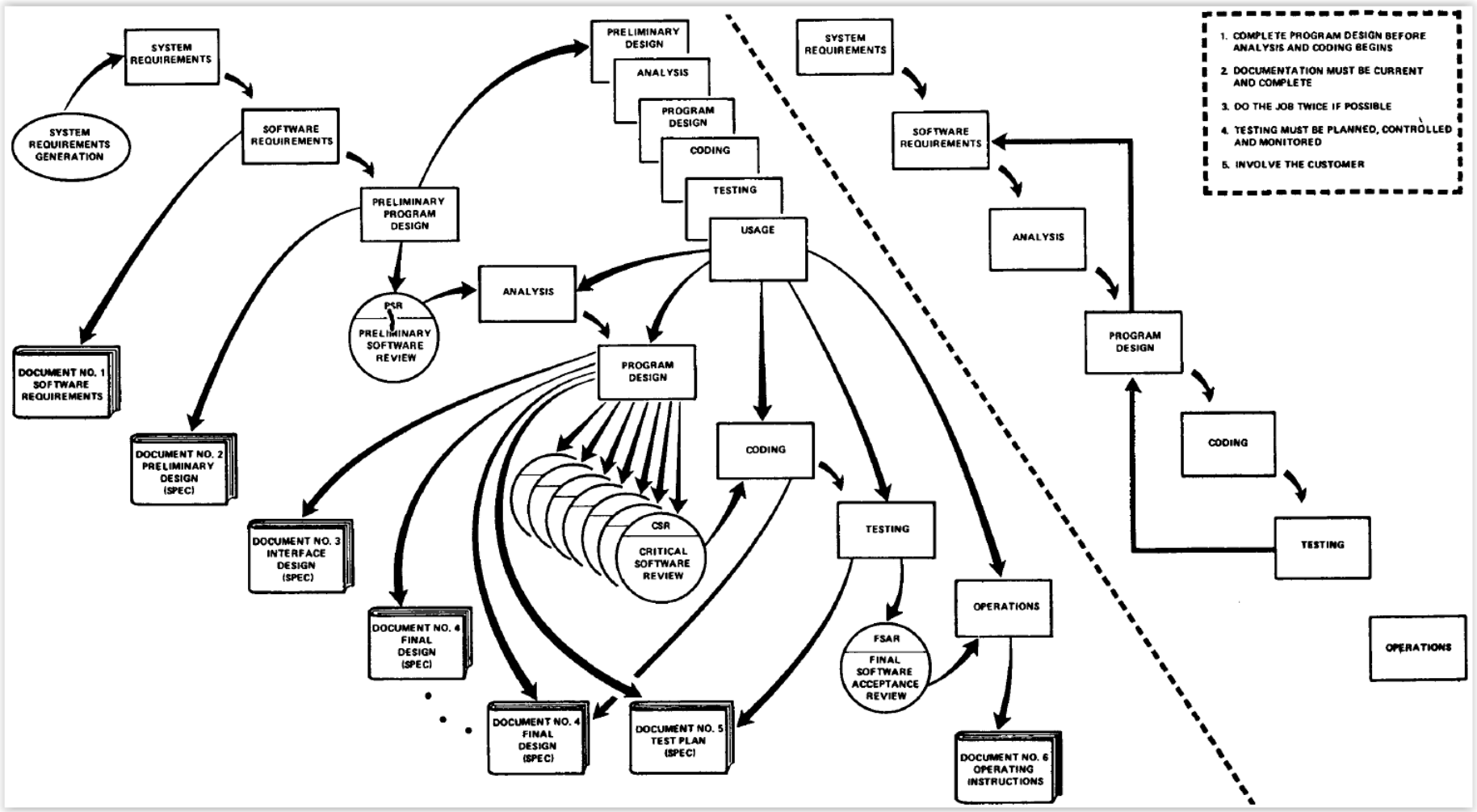

Рисунок 10 Итоговая схема