# Этапы разработки проекта: реализация, тестирование, эксплуатация и сопровождение

## Реализация

Трудно давать советы по реализации кода модулей, так как каждый разработчик имеет какие-то привычки и свой стиль разработки кода. При реализации проекта важно координировать группу (группы) разработчиков. Все разработчики должны подчиняться жестким правилам контроля исходных тестов. Группа разработчиков, получив технический проект, начинает писать код модулей, и в этом случае основная задача состоит в том, чтобы уяснить спецификацию. Проектировщик указал, что необходимо сделать, а разработчик определяет способы выполнения.

На этапе разработки осуществляется тесное взаимодействие проектировщиков, разработчиков и групп тестеров. В случае интенсивной разработки тестер буквально «пристегивается» к разработчику, фактически являясь членом группы разработки.

Проектировщик на данном этапе выполняет функции «ходячего справочника», поскольку постоянно отвечает на вопросы разработчиков, касающиеся технической спецификации.

Чаще всего на этапе разработки меняются интерфейсы пользователя. Это обусловлено в том числе и тем, что модули периодически демонстрируются заказчику. Существенно могут меняться и запросы к данным.

Следует отметить, что для сборки всего проекта должно быть выделенное рабочее место. Именно эти модули передаются на тестирование. Взаимодействие тестера и разработчика без централизованной передачи частей проекта допустимо, но только в случае, если необходимо срочно проверить какую-то правку. Очень часто этап разработки и этап тестирования взаимосвязаны и идут параллельно. Синхронизирует действия тестеров и разработчиков система bug tracking.

При разработке должны быть организованы постоянно обновляемые хранилища готовых модулей проекта и библиотек, которые используются при сборке модулей. Желательно, чтобы процесс обновления хранилищ контролировал один человек. Одно из хранилищ должно быть предназначено для модулей, прошедших функциональное тестирование,  а другое - для модулей, прошедших тестирование связей. Первое из них - это черновики. Второе - то, из чего уже можно собирать дистрибутив системы и демонстрировать его заказчику для проведения контрольных испытаний или сдачи каких-либо этапов работ.

Документация создается в течение всего процесса разработки. Как только модуль прошел тестирование связей, его можно описывать в документации. В случае если модули изменяются часто, к описанию приступают только тогда, когда модуль становится более или менее стабильным.

### Обработка результатов проектирования

На этапе разработки, как правило, еще раз проверяется атомарность функций, а также отсутствие их дублирования.

Желательно, чтобы на этапе проектирования уже была построена матрица «функции-сущности». Это фактически формализованное представление того, что фирма пытается сделать (функции) и какую информацию требуется обработать для достижения результата (сущности). Подобная матрица позволяет проверить следующие моменты:

* имеет ли каждая сущность конструктор - функцию, создающую экземпляры сущности (create);
* есть ли ссылки на данную сущность, то есть используется ли где-либо данная сущность (references);
* имеют ли место изменения данной сущности (update);
* имеет ли каждая сущность деструктор - функцию, которая удаляет экземпляры сущности (delete).

Часто роль деструктора выполняет комплект программ архивирования данных. Нередко в информационных системах информацию просто накапливают. Это допустимо лишь в том случае, если в течение всего периода накопления информации (а фактически в течение всей жизнедеятельности информационной системы) характеристики ее производительности удовлетворяют требованиям заказчика. На практике это чрезвычайно редкое стечение обстоятельств. Связано это в основном с ростом обрабатываемых объемов информации. Следует отметить, что надеяться в этом случае только на мощность СУБД или аппаратного обеспечения нельзя, так как подобные экстенсивные методы повышения производительности дают низкий расчетный прирост скорости. Фактически задача реагирования системы или отдельных ее частей на рост объема обрабатываемых данных является наиболее вероятной задачей тестирования. В таком случае группа тестирования создает модуль генерации (пусть даже абстрактных) данных, выбирается набор запросов, для которых скоростные характеристики критичны, далее производятся замеры и строится зависимость скорости выполнения от объема данных для каждого из запросов. Такое простое действие позволит избежать серьезных ошибок и в проектировании, и в реализации информационной системы.

Спецификация модулей должна быть выполнена еще на этапе проектирования, чему в реальных проектах зачастую просто не придают значения. И напрасно - ведь из-за непродуманной реализации модулей любые достоинства схемы базы данных могут быть утрачены. Так, пренебрегая спецификациями модулей, вы рискуете заложить в информационную систему:

* неконтролируемый рост объемов данных;
* потоки запросов с изначально высокой вероятностью конфликта или потоки запросов, которые будут выполняться «вечно» (попытка выполнить поток, обнаружение конфликта и откат всех действий, новая попытка и т.д.) из-за конфликтующих с ними потоков;
* смешивание системных и интерфейсных модулей;
* дублирование модулей;
* ошибки в размещении бизнес-логики;
* отсутствие реализации или неполная реализация требуемых заказчиком функций системы.

Это далеко не полный список проблем, которые будут обнаружены или на этапе комплексного тестирования, или при вводе системы в эксплуатацию, а может быть, даже в процессе эксплуатации системы (когда начнут реально использоваться модули).

Кроме того, отсутствие спецификаций модулей не позволит точно оценить сложность каждого модуля и, как следствие, определить последовательность создания модулей и правильно распределить нагрузку персонала. Обычная ситуация в подобном случае  -  «кто-то кого-то ждет», при этом процесс создания информационной системы стоит на месте.

### Системные модули

Часто приходится рассматривать большое количество обслуживающих или вспомогательных процессов, которые непосредственно не связаны со сформулированной бизнес-функцией. Как правило, это системные функции, имеющиеся в любой информационной системе, такие как:

* диспетчер очередей или планировщик заданий;
* диспетчер печати;
* средства доступа к данным и создания нерегламентированных запросов (часто это генераторы отчетов);
* управление каталогами и иными ресурсами файловой системы;
* автоматическое резервное копирование;
* автоматическое восстановление после сбоя системы;
* средства регламентирования доступа пользователей к системе (состоящие из средства создания пользователей и средства назначения им привилегий);
* средство настройки среды для пользователя информационной системы;
* средство изменения пользователем своих настроек (в том числе и пароля);
* средство управления приложениями;
* среда администратора информационной системы.

Часть этих функций должна выполнять операционная система, но если она будет работать в неоднородной среде, то нет гарантии, что пользователям придется по вкусу наличие различных интерфейсов в разных операционных системах. В идеале все приложения-клиенты должны работать в одной операционной системе, однако на практике разработчикам часто приходится сталкиваться с целым «зоопарком» различных рабочих станций у заказчика - итогом нескольких попыток автоматизировать бизнес. Цель разработчика - довести систему до максимально однородного состояния либо сделать похожими хотя бы рабочие места конечных пользователей.

Задача создания информационной системы в разнородной среде существенно повышает требования к разработчикам кода и к выбираемому средству разработки. Особенно это касается разработки системных модулей. Следует уделить внимание модулям, реализация кода которых зависит от операционной системы. Подобные модули должны быть выделены отдельно для каждой из операционных систем в группы, например Win98, WinNT и т.д. Модули каждой из групп должны иметь строгие интерфейсы обмена - данные, которые они передают и получают, строго определены, любое отклонение от спецификации наказуемо. Ни один из модулей вне этой группы не может использовать никаких других вызовов, кроме интерфейсов обмена. Таким образом модули, зависящие от операционнй системы, изолируются от других модулей.

Вообще говоря, практика изолирования системных модулей посредством строгой регламентации их интерфейсов обмена существенно минимизирует затраты по исправлению ошибок и поддержке системы. Кроме того, это облегчает и тестирование, а именно детектирование ошибок и их отладку. Другая сторона вопроса - требования к коду интерфейса обмена системных модулей резко повышаются. Это то, что отлаживается в первую очередь и должно работать очень четко.

### Средства мониторинга информационной системы

Если информационная система велика, то следует рассмотреть задачу ее администрирования с одной рабочей станции. Необходимо заботиться не только о конечном пользователе информационной системы, но и о персонале, который будет ее обслуживать. Особое внимание следует уделить мониторингу критических участков информационной системы, поскольку сбой зачастую проще предотвратить, чем исправлять его последствия. Мониторинг относится к тем задачам, о необходимости решения которых заказчик, как правило, не задумывается и которые обычно отсутствуют и в аналитическом исследовании, и даже при проектировании. Потребность в средствах мониторинга становится очевидной лишь на этапе ввода системы в эксплуатацию, причем потребность эта тем выше, чем сложнее система и чем больше в ней критических участков.

Разработчикам и проектироващикам следует проводить оценку сложности системы. Если принимается решение о написании комплексного средства администрирования и мониторинга, не предусмотренного техническим заданием, то в этом случае следует менять техническое задание, а не идти на поводу у заказчика. В сложной системе отслеживать критические процессы все равно придется. Внедрять подобные средства в уже готовую систему очень сложно, поскольку исходные данные мониторы часто получают от системных модулей достаточно низкого уровня. Без изменений схемы базы данных здесь тоже вряд ли можно будет обойтись, и нет никакой гарантии, что подобное изменение не ухудшит производительность системы.

Разработка мониторов - это довольно специфический класс задач: с одной стороны, они должны обрабатывать достаточный объем информации, с другой - не должны существенно влиять на работу других компонентов информационной системы. Это заставляет разработчиков с особой тщательностью подходить к проектированию мониторов и очень аккуратно писать код их модулей.

### Интерфейсы

Интерфейсы конечного пользователя - это то, что заказчик критикует в наибольшей степени, в силу того что именно эти части информационной системы он может более или менее квалифицированно оценить - обычно только их он и видит. Это означает, что интерфейсы являются наиболее часто изменяемым элементом информационной системы именно на этапе реализации.

Часто изменяемый компонент (компоненты) информационной системы следует изолировать от редко изменяемых компонентов,  чтобы одни изменения не влекли за собой другие. Один из приемов подобной изоляции - изоляция запросов  к данным от интерфейса следующим образом:

* каждый из запросов кодируется идентификатором или «закрывается» определенной системной функцией;
* разработчик интерфейса не знает о запросе к данным ничего, кроме параметров атрибутов выборки - их типа и, возможно, количества строк в выборке;
* обработка ошибок в запросах данных представляет собой отдельный модуль;
* обработка ошибок в интерпретации результата запроса также представляет собой отдельный модуль.

При обработке результатов запросов данных следует также особое внимание уделить вопросам соответствия типов включающего языка и СУБД, в том числе вопросам точности числовых типов, так как представление их у разных СУБД существенно различается. Кроме того, обратите внимание на запросы к данным, которые используют функции, зависящие от операционной системы, например функции работы с  байтами и словами значения атрибута (например, на Intel и SUN SPARC эти функции будут работать по-разному). Типы данных могут быть приведены или явно в запросе функциями приведения cast и встроенными в СУБД функциями, или в функции прикладной программы. Не для всех СУБД неявное преобразование типов дает один и тот же результат, поэтому если информационая система использует данные из нескольких баз данных под управлением разных СУБД, то неявных преобразований типов лучше избегать.

Следует также установить достаточно жесткие правила для внешнего вида интерфейсов пользователя. Должно создаваться впечатление единого стиля для всех компонентов информационной системы.

### Версии базы данных

Первую версию базы данных проекта в большинстве случаев создают достаточно быстро - это реализация полностью нормализованной структуры, которую получают на этапе анализа. Основным назначением этой базы данных является обеспечение макетирования, демонстрационных показов, некоторых экспериментов разработчиков и проектировщиков.

Скрипты создания базы данных и заполнения ее стартовыми данными - это тоже исходный код информационной системы, и на него распространяются правила контроля версий. Следует отметить, что поддерживать версии базы даных на уровне скриптов все же проще, чем на уровне средств выгрузки и загрузки данных, предоставляемых самой СУБД, так как в подавляющем большинстве случаев подобные средства не могут предоставить несколько простых, но необходимых функций:

* проконтролировать, какие объекты данных и данные имеют место в объектах загрузки A и B, и загрузить в базу данных только «разницу» A и B (произвести обновление версии);
* проконтролировать, не конфликтуют ли изменения, имеющие место в объектах выгрузки C и D, по сравнению с объектом выгрузки A (произвести слияние версий).

CASE-инструменты имеют средства контроля версий схемы базы данных, некоторые имеют настройки, позволяющие также контролировать и стартовые данные. Это дает возможность использовать указанные средства для обеспечения контроля версий базы данных.

Контроль версий исходного кода триггеров, хранимых процедур надежнее осуществлять путем использования той же системы контроля версий, что принята для хранения исходных текстов самого проекта.

### Размещение логики обработки

Одим из важных вопросов проектирования является способ размещения бизнес-логики обработки данных: размещать ее (и какую часть) либо на сервере в виде хранимых процедур, пакетов, триггеров, иных ограничений целостности непосредственно на сервере баз данных, либо в виде функций на клиенте (в составе ПО клиента). Местонахождение правил интерфейса и правил данных задано точно: первые всегда размещены на клиенте, вторые - на сервере. Правила бизнес-логики в современных СУБД могут быть размещены как на клиенте, так и на сервере. Рассмотрим один из примеров простейшего бизнес-правила:

* Значение в поле экранной формы вводится пользователем, а не выбирается из списка, но набор допустимых значений строго ограничен (например, два или три различных значения).

С одной стороны, пользователь требует немедленной реакции системы на ошибку ввода данных, с другой - недопустимы значения в поле базы данных, отличные от заданных (двух или трех). На самом деле в этой ситуации  должны быть реализованы два правила. Правило данных в этом случае будет организовано в виде ограничения check, а правило интерфейса, запрещающее вводить значения, отличные от заданных, будет в точности повторять правило даных, но будет реализовано на уровне интерфейса пользователя. Казалось бы, реализация формы со списком в этом случае является идеальным решением, но большинство операторов предпочитают именно набор в форме, особенно если длина вводимого значения невелика. Формы с большим количеством списков достаточно трудны для обработки конечными пользователями. В случае набора значений в форме следует также позаботиться о приведении регистров строк символов (там, где регистр не существенен) к верхнему или нижнему регистру, на уровне интерфейса прикладной программы.

### Шаблоны

Использование шаблонов и библиотек для построения «похожих» модулей - достаточно распространенная практика. Что использовать в этом случае - объекты и классы или библиотеки - решает конкретная группа разработчиков. Диктовать способ разработки в большинстве случаев бессмысленно, потому что разработчик пишет код так, как умеет или как привык. Эти моменты обычно контролирует руководитель проекта.

В любом проекте запрещается копирование кода, поскольку это ведет к возникновению различных версий одного и того же кода в разных фрагментах  прикладной программы и, как следствие, к сложно детектируемым и исправляемым ошибкам. Следует установить жесткое правило: используется вызов функции, а не его копия в коде; любое отклонение от данного правила наказуемо.

## Тестирование

Как было сказано выше, группы тестирования могут привлекаться уже на ранних стадиях разработки проекта. Собственно комплексное тестирование действительно следует выделять в отдельный этап разработки. В зависимости от сложности проекта тестирование и исправление ошибок могут занимать треть, половину и больше времени разработки всего проекта.

Чем сложнее проект, тем больше будет потребность в автоматизации системы хранения ошибок - bug tracking. Подобная система обеспечивает следующие функции:

* хранение сообщения об ошибке (с обязательной информацией о том, к какому компоненту системы относится ошибка, кто ее нашел, как ее воспроизвести, кто отвечает за ее исправление и когда она должна быть исправлена);
* система уведомления о появлении новых ошибок, об изменении статуса известных в системе ошибок (как правило, это уведомления по электронной почте);
* отчеты об актуальных ошибках по компонентам системы, по интервалам времени, по группам разработчиков и разработчикам;
* информация об истории ошибки (в том числе отслеживание похожих ошибок, отслеживание повторного возникновения ошибки);
* правила доступа к ошибкам тех или иных категорий;
* интерфейс ограниченного доступа к системе bug tracking для конечного пользователя информационной системы, который используется как интерфейс обмена информацией между пользователем и службой технической поддержки системы.

Подобные системы снимают множество организационных проблем, в частности вопросы автоматического уведомления об ошибках.

Собственно тесты систем можно разделить на несколько категорий:

* автономные тесты модулей - используются уже на этапе разработки компонентов системы и позволяют отслеживать ошибки отдельных компонентов;
* тесты связей компонентов системы - используются и на этапе разработки, и на этапе тестирования и позволяют отслеживать правильность взаимодействия и обмена информацией компонентов системы;
* системный тест - является основным критерием приемки системы. Как правило, это группа тестов, включающая в себя и автономные тесты, и тесты связей и модели. Данный тест должен воспроизводить работу всех компонентов и функций системы, его основная цель - внутренняя приемка системы и оценка ее качества;
* приемо-сдаточный тест - используется при сдаче системы заказчику. Здесь разработчики часто занижают требования к системе по сравнению с системным тестом, и в общем-то понятно, почему это оправданно;
* тесты производительности и нагрузки - входят в системный тест, но достойны отдельного упоминания, так как именно эта группа тестов является основной для оценки надежности системы.

В тесты каждой группы обязательно входят тесты моделирования отказов. Здесь проверяется реакция компонента, группы компонентов, системы в целом на отказы следующего типа:

* отказ отдельного компонента информационной системы;
* отказ группы компонентов информационной системы;
* отказ основных модулей информационной системы;
* отказ операционной системы;
* «жесткий» сбой (отказ питания, жестких дисков).

Эти тесты позволяют оценить качество подсистемы восстановления корректного состояния информационной системы и служат основным источником информации для разработки стратегий предотвращения негативных последствий сбоев при промышленной эксплуатации. Как правило, это тот класс тестов, которым разработчики пренебрегают, а затем борются с последствиями сбоев на промышленной системе.

Еще одним важным моментом программы тестирования информационных систем является наличие генераторов тестовых данных. Они используются для проведения как тестов функциональности системы, так и тестов надежности системы, а также тестов производительности системы. Задача оценки характеристик зависимости производительности информационной системы от роста объемов обрабатываемой информации не может быть решена без генераторов данных.

## Эксплуатация и сопровождение

Опытная эксплуатация перекрывает процесс тестирования. Как правило, система вводится в эксплуатацию не полностью, постепенно.

Ввод в эксплуатацию проходит по крайней мере три фазы:

1. первоначальная загрузка информации;
2. накопление информации;
3. выход на проектную мощность.

Первоначальная загрузка информации инициирует довольно узкий круг ошибок - в основном это проблемы рассогласования данных при загрузке и собственные ошибки загрузчиков, то есть то, что не было отслежено на тестовых данных. Подобные ошибки должны быть исправлены как можно быстрее. Не поленитесь поставить отладочную версию системы (если, конечно, вам позволят развернуть весь комплекс сопровождающего отладку информационной системы ПО на месте). Если отладку «на живых» данных производить невозможно, то придется моделировать ситуацию, причем быстро. Здесь требуются очень квалифицированные тестеры.

В период накопления информации  проявится наибольшее количество ошибок, допущенных при создании информационной системы.   Как правило, это ошибки, связанные с многопользовательским доступом. Часто на этапе тестирования таким ошибкам не уделяется должного внимания - видимо, из-за сложности моделирования и дороговизны средств автоматизации процесса тестирования информационной системы в условиях многопользовательского доступа . Некоторые ошибки исправить будет довольно сложно, так как они являются ошибками проектирования. Ни один самый хороший проект от них не застрахован. Это значит, что на всякий случай надо резервировать время на локализацию и исправление таких ошибок.

Вторая категория исправлений связана с тем, что пользователя не устраивает интерфейс. Здесь не всегда нужно выполнять абсолютно все пожелания пользователя, иначе процесс ввода в эксплуатацию не кончится никогда.

В период накопления информации можно столкнуться со знаменитым «упала база». При самом плохом раскладе окажется, что СУБД не выдерживает потока информации. При хорошем - просто параметры конфигурации неверны. Первый случай опасен, так как повлиять на производителя СУБД довольно сложно, а заказчик очень не любит ссылок на службу технической поддержки СУБД. Решать проблему отказа СУБД придется не производителю, а вам - менять схему, снижать поток запросов, менять сами запросы; в общем - вариантов много. Хорошо, если время восстановления базы вписывается в запланированное.

Выход системы на проектную мощность при удачном стечении обстоятельств - это исправление ряда мелких ошибок, и изредка - ошибок серьезных.

## Другие подходы к разработке приложений

Как правило, конечные пользователи и руководство полагают, что процесс проектирования не дал никаких результатов, поскольку  отсутствуют готовые компоненты, которые можно было бы «пощупать». Зачастую заказчик настаивает на досрочном проведении этапа реализации проекта, для того чтобы как можно быстрее получить какой-то результат и продемонстрировать его. В таком случае существует большой соблазн выбрать ускоренную разработку приложений (УРП) или совместную разработку приложений (СРП). Подобные методы предусматривают разработку рабочего прототипа с последующей демонстрацией его пользователям. Пользователи отмечают, что им нравится, а что - нет. Проектировщик дорабатывает прототип с учетом сделанных замечаний, после чего снова демонстрирует то, что получилось. И так далее. Процесс повторяется до тех пор, пока пользователям не понравится то, что они видят, а прототип не станет рабочим приложением. Обычно устанавливается лимит времени и количество итераций, иначе пользователи будут совершенствовать прототип вечно. Теоретически это позволяет получить ту систему, которая требуется пользователям. На практике подобный подход к разработке приложений сопряжен с серьезными проблемами.

* Все внимание сконцентрировано на экранных формах, а то, что касается правил обработки данных и системных функций, остается за кадром. Есть соблазн начать работу с отчетов, в то время как отчет является не стартовым, а производным продуктом информационной системы.
* Пользователи полагают, что если вариант прототипа согласован, то модуль готов. На самом деле это может быть всего лишь картинка с набором «заглушек» для вызовов системных функций и взаимодействия с другими модулями.
* Модули проектируются изолированно друг от друга (наверное, большинство из вас сталкивались с бухгалтерскими программами, где каждый АРМ является автономным и функции часто дублируются). Следствием этого являются противоречия модулей, дублирование функций и данных, что может быть выявлено только при тестировании комплекса модулей.
* Функциональные возможности наращиваются параллельно в нескольких направлениях, значит, структура базы данных должна контролироваться жестко. При УРП схема базы данных превращается в свалку, где таблицы «лепятся» наскоро, в результате чего имеет место набор противоречивых и дублирующихся данных.
* Документация при использовании метода УРП, как правило, отсутствует, а вернее, о необходимости документировать систему забывают, поскольку создается иллюзия, что пользователь и без того понимает, что происходит. Когда же приложение начинает работать не так, как предполагает пользователь, возникает масса проблем.
* Обработка исключительных ситуаций для каждого модуля производится своя.
* Целостная система, как правило, не получается, скорее всего, это будет некий набор автоматизированных рабочих мест, наскоро связанных между собой.

Методы УРП и СРП можно использовать далеко не всегда, а лишь в том случае, если:

* объем проекта и требования бизнеса четко определены, не изменяются, а сам проект невелик;
* проект не зависит от других средств автоматизации бизнеса, количество внешних интерфейсов, с которыми придется иметь дело, ограниченно;
* система ориентирована на экранные формы, обработка данных и системные функции составляют незначительную часть, удобство экранных форм входит в пятерку важнейших факторов успеха проекта;
* пользователи имеют высокую квалификацию и априори положительно оценивают идею создания нового ПО.

Тем не менее методом УРП лучше разрабатывать небольшие и, желательно, автономные части проекта.

В настоящее время предпринята попытка представить еще один способ быстрого написания проекта - метод экстремального программирования.  Ниже будут рассмотрены принципы данного подхода.

Этап планирования (planning game). На основании оценок, сделанных программистами, заказчик определяет функциональные возможности и срок реализации версий системы. Программисты реализуют только те функции, которые необходимы для возможностей, выбранных на данной итерации.

В результате такого решения «за кадром» остается развитие системы, вследствие чего при разработке возникает необходимость строить «заглушки» и переписывать код. Непонятно, почему срок реализации определяет заказчик, ведь на самом деле это прямая обязанность группы проектировщиков. Заказчик, вообще говоря, может лишь выразить свои пожелания по поводу сроков («хочу, чтобы к такому-то числу»), но определить срок может только проектировщик («выполнимо не меньше чем за такое-то время»).

Частая смена версий (small releases). Систему запускают в эксплуатацию уже через несколько месяцев после начала реализации, не дожидаясь окончательного разрешения всех поставленных проблем. Выпуск новых версий может происходить с периодичностью от ежедневного до ежемесячного.

Все хорошо, кроме одного: протестировать за такой срок более или менее сложный компонент невозможно. Заказчик фактически выступает в роли бета-тестера. В этом случае он может видеть, что разработчики трудятся и даже ошибки исправляют. Однако возникают резонные вопросы: стоит ли посвящать заказчика в рабочий процесс и нужно ли ставить эксперименты на рабочей системе? В дополнение к сказанному необходимо отметить, что подобный принцип вряд ли может быть реализован для частей проекта, которые требуют работы в режиме 24x7.

Метафора (metaphor). Общий вид системы определяется при помощи метафоры или набора метафор, над которыми совместно работают заказчик и программисты.

С одной стороны, этот постулат кажется неплохим, а с другой - имеет ли смысл посвящать заказчика во внутренние дела группы разработчиков? То, что касается общего вида (интерфейсы, отчеты и т.п.), действительно может находиться в компетенции заказчика,  но когда речь идет об особенностях реализации тех или иных компонентов, заказчик вряд ли может быть полезен из-за отсутствия у него необходимых знаний.

Простой проект (simple design). В каждый момент времени разрабатываемая система выполняет все тесты и поддерживает все взаимосвязи, определяемые программистом, не имеет дубликатов кода и содержит минимально возможное количество классов и методов. Это правило кратко можно выразить так: «Каждую мысль формулируй один и только один раз».

Эта мысль тоже хороша, но она не вполне согласуется с принципом быстрого написания кода. Может быть, стоит все-таки сначала подумать, как делать тот или иной модуль, группу модулей, и лишь потом заняться написанием кода?

Тесты (tests). Программисты постоянно пишут тесты для модулей (unit tests). Собранные вместе, эти тесты должны работать корректно. Для этапов в итерации заказчики пишут функциональные тесты (functional tests),  которые также должны работать правильно. Однако на практике это не всегда достижимо. Чтобы принять правильное решение, необходимо понять, во сколько обойдется сдача системы с заранее известным дефектом, и сравнить это с ценой задержки на исправление дефекта.

При написании тестов самими программистами (особенно в условиях сверхурочных работ) эти тесты не полнофункциональны, и уж тем более не учитывают особенностей многопользовательской работы. На более продвинутые тесты у разработчиков обычно не хватает времени. Можно, конечно, построить систему разработки так, что всем будут заниматься одни и те же люди, но все-таки не стоит превращать проект в аналог телепередачи «Сам себе режиссер». К сказанному необходимо добавить, что тестирование системы вовсе не исчерпывается тестами компонентов (units); не менее важны тесты взаимодействия между ними, это же относится и к тестам надежности работы. И тем не менее метод экстремального программирования не предусматривает создания тестов данного класса. Это объясняется тем, что сами такие тесты могут представлять достаточно сложный код (особенно это касается тестов-имитаторов реальной работы системы). В данной технологии также никак не учитывается еще один важный класс тестов - тесты поведения системы при росте объемов обрабатываемой информации. При высокой скорости изменения версий выполнить такой тест технологически невозможно, поскольку его проведение требует стабильного и неизменного кода проекта, например, в течение недели. Подобные сроки, вообще говоря, не гарантируются из-за частой смены версий. В таком случае придется или приостанавливать разработку компонентов, или на время проведения теста создавать параллельную версию проекта, которая будет сохраняться неизменной, тогда как вторая при этом будет изменяться. Потом нужно будет выполнять процесс слияния кода. Но в этом случае тест придется создавать снова, так как методы экстремального программирования просто не предусматривают разработку средств, позволяющих прогнозировать поведение системы при тех или иных изменениях.

Переработка системы (refactoring). Архитектура системы постоянно эволюционирует. Текущий проект трансформируется, при этом гарантируется правильное выполнение всех тестов.

Вот тут-то и начинается самое интересное. Экстремальное программирование исходит из того, что переделать всегда можно, причем без особых затрат. Однако практика свидетельствует об обратном.

Программирование в паре (pair programming). Весь код проекта пишут два человека, которые используют одну настольную систему.

Возникает вопрос: кто-нибудь видел двух  совершенно одинаковых программистов, каждый из которых к тому же в конце рабочего дня успевал бы писать документацию для напарника?  Можно ли найти таких программистов-близнецов, согласных во всем?

А главное, зачем нужна такая пара программистов? Причина, в общем-то, простая: не все выдерживают навязываемый при экстремальном программировании высокий темп работ, неизбежен отток персонала. Подобная пара может дать некую страховку - если уволится один, то, может быть, второй доведет дело до конца. Правда, оставшийся попадет в еще более жесткие временные рамки - ведь объем работ останется прежним же, а дублера уже не будет, по крайней мере какое-то время. Далее следует естественный процесс передачи информации новому дублеру, что опять-таки требует времени. И так без конца.

Непрерывная интеграция (continuous integration). Новый код интегрируется в существующую систему не позднее, чем через несколько часов. После этого система вновь собирается в единое целое и прогоняются все тесты. Если хотя бы один из них не выполняется корректно, внесенные изменения отменяются.

Этот постулат предоставляется по меньшей мере спорным, поскольку непонятно, кто будет исправлять ошибки, причем не только локальные, но и наведенные неправильным кодом. Ведь проведение комплексных тестов не предполагается на данном этапе, кроме того, изменения остаются даже в том случае, когда ошибка детектирована. В то же самое время метод экстремального программирования не предусматривает наличия системы отслеживания ошибок.

Коллективное владение (collective ownership). Каждый программист имеет возможность в любое время усовершенствовать любую часть кода в системе, если сочтет это необходимым.

Вам это анархию не напоминает? Как в этом случае искать автора изменений? Встречал ли кто-либо при разработке большого проекта такого «на все руки доку»  и сколько подобный «умелец»  сумел бы продержаться на своем рабочем месте? Правильно, не слишком долго.

Заказчик с постоянным участием (on-site customer). Заказчик, который в период работы над системой находится в команде разработчиков.

Это, конечно, хорошо, но непонятна цель: то ли посвятить заказчика в суть дела, то ли сделать его соавтором? Вряд ли только у заказчика найдется столь высококвалифицированный специалист.

40-часовая неделя (40-hour weeks). Объем сверхурочных работ не может превышать по длительности одну рабочую неделю. Даже отдельные случаи сверхурочных работ, повторяющиеся слишком часто, являются сигналом серьезных проблем, которые требуют безотлагательного решения.

Как показывает практика применения экстремального программирования (несмотря на целый ряд положительных примеров, приводимых сторонниками данного метода), сверхурочные при таком подходе - это правило, а не исключение, и борьба с проблемами в данном случае - явление постоянное. Усиливается она в период замены текущей сырой версии продукта очередной, опять же сырой, версией. Заказчик, посвященный в процесс, испытывает все прелести проявления ошибок работы системы на себе. Как вы думаете, надолго ли хватит у заказчика терпения при таком положении дел? Ему ведь надо, чтобы система работала...

Открытое рабочее пространство (open workspace). Команда разработчиков располагается в большом помещении, окруженном комнатами меньшей площади. В центре рабочего пространства устанавливаются компьютеры, на которых работают пары программистов.

Причем все это, судя по предыдущим принципам, должно располагаться на территории заказчика, раз он весьма активно привлекается к процессу разработки. Возникает вопрос: реально ли столь удачное стечение обстоятельств?

Не более чем правила (just rules). Члены коллектива, работающего по технологии экстремального программирования, обязуются выполнять изложенные правила. Однако это не более чем правила и команда может в любой момент поменять их, если ее члены достигли принципиального соглашения по поводу внесенных изменений.

Может быть, в конце концов и будет выработано одно полезное правило: «сначала подумай, потом сделай». В этом случае мы будем иметь схему, весьма похожую на «водопад». Почему-то сторонники экстремального программирования убеждены, что при использовании «водопада» и его клонов цикл разработки обязательно должен быть длинным. Непонятно, чем обусловлена такая уверенность. Ведь не запрещено дробить проект на этапы. Почему-то считается, что планирование обязательно будет одноразовым и неизменным, хотя на самом деле это не соответствует истине, в том числе и в случае «водопада».

В итоге мы получаем метод, потенциально обладающий высокой адаптируемостью к сильно изменяющимся требованиям к проекту, но в то же время не свободный от ряда серьезных недостатков. Последнее обстоятельство не позволяет рекомендовать данный метод к применению для проектов, требующих высокой или как минимум достаточной надежности работы.