Содержание

Введение

Данная работа представляет собой экспериментальное исследование, направленное на изучение природы основных методологических проблем, возникающих при внедрении ERP систем на отечественных предприятиях и практическое их решение.
В настоящее время на рынке программного обеспечения продолжается рост популярности ERP систем. Все больше и больше предприятий в различных отраслях внедряют у себя ERP системы. При этом возникает масса проблем, связанных с внедрением этих систем. К ним относятся как проблемы, связанные с изменением управления бизнесом во время и после внедрения систем, так и технологические проблемы в самом процессе внедрения. Некоторым из них и посвящена данная работа "Внедрение ERP систем на отечественных предприятиях с использованием методологии ASAP".
В наше время технологии стремительно развиваются: постоянно появляются новые, более совершенные, заменяя собой устаревшие. И за последние тридцать лет достигли больших высот. В таком случае, почему проекты по разработке и внедрению не самого сложного по своей сути программного обеспечения, каким являются ERP системы, до сих пор сталкиваются с теми же проблемами, что и

тридцать лет назад? Видимо, ответ на этот вопрос следует искать не в технологии производства и внедрения программного обеспечения, а в методологии процесса этого производства и внедрения. Таким образом, основным предметом исследования настоящей работы является изучение именно методологических проблем внедрения ERP систем на отечественных предприятиях, как наиболее важных и актуальных в этой отрасли в нашей стране и во всем мире.

Не смотря на то, что рынок ERP систем начал развиваться в нашей стране относительно недавно, на нем успели появиться свои лидеры - наиболее популярные у нас в стране ERP системы. Одной из таких наиболее популярных систем является система mySAP ERP 2005 SR2 - один из новейших продуктов известной во всем мире немецкой компании SAP AG. Интересно отметить, что ERP системы SAP, как и прочие продукты этой компании, поставляются вместе с собственной методологией их внедрения. Эта методология называется AcceleratedSAP (ASAP). Сам факт наличия методологии, специально разработанной компанией-производителем ERP системы для ее внедрения, вызывает интерес. Поэтому именно эта методология в данной аттестационной работе является объектом изучения методологических проблем внедрения ERP систем на отечественных предприятиях.

В мире существует множество различных, иногда противоположных, методологий разработки и внедрения программного обеспечения, основанных на лучших разработках крупнейших в этой области ученых и многолетнем опыте применения этих разработок на практике с использованием специализированных инструментов. Именно изучение этих разработок и методологий и сопоставление с ними методологии ASAP и выбрано в этой работе основным методом изучения методологических проблем внедрения ERP систем на отечественных предприятиях с использованием методологии ASAP. Любая ERP система является характерным конкретным экземпляром более абстрактного понятия "программное обеспечение". Поэтому такое сопоставление корректно.

Для раскрытия данной темы необходимо решить ряд вопросов:

· Почему проблемы, возникавшие тридцать лет назад, до сих пор актуальны?

· Что является источником этих проблем?

· Как подойти к разработке и внедрению ERP систем, минуя эти проблемы?

· Какой должна быть избавляющая от проблем методология разработки и внедрения ERP систем?

Чтобы получить ответы на эти вопросы, необходимо решить ряд задач:

· Раскрыть сущность программного обеспечения: что это такое, кто его создает и как;

· Рассмотреть понятие методологии процесса создания и внедрения программного обеспечения: изучить основные подходы к методологиям управления и оценки процесса;

· Классифицировать и сопоставить основные методологии, выбрать из них наиболее подходящую для внедрения ERP систем;

· Изучить методологию ASAP и, сопоставив ее с выбранной методологией, выявить ее сильные и слабые стороны;

· Выявить существенные методологические недостатки ASAP, вызывающие

методологические проблемы при использовании ASAP для внедрения ERP систем компании SAP AG на отечественных предприятиях, предложить пути решения этих проблем.

Итак, попытаемся решить эти задачи и разобраться в поставленных вопросах, изучив опубликованные в нашей стране и за рубежом очерки и книги ведущих специалистов в области разработки и внедрения программного обеспечения, а также методологический материал AcceleratedSAP и других методологий.

1. Современные методологические проблемы разработки и внедрения программного обеспечения ERP систем

1.1 Философия программного обеспечения

Философия одного века - это здравый смысл следующего (Генри Уорд Бичер).

Прежде, чем приступить к исследованиям методологических проблем разработки и внедрения программного обеспечения ERP систем века нынешнего, следуя мудрости американского религиозного деятеля XIX века, обратимся к мыслям теоретиков и практиков века прошлого. Прошлый век подарил нам массу теоретических разработок и практических изобретений в области вычислительной техники и программного обеспечения, применение которых в нынешнем веке порождает множество методологических проблем. Поиски здравого смысла современности следует начать во второй половине XX века. С точки зрения философии, "программное обеспечение" - это очень молодая, но очень интересная категория. Изучим программное обеспечение как абстрактную философскую категорию, чтобы раскрыть ее сущность, скрытую за множеством конкретных деталей. Рассмотрим общие подходы к решению трех основных вопросов:

· ЧТО создается и внедряется в качестве программного обеспечения?

· КАК создается и внедряется программное обеспечение?

· КТО создает и внедряет программное обеспечение?

К сожалению, в третьей четверти XX века, когда закладывались основы разработки и внедрения программного обеспечения, как в нашей стране, так и во всем мире, различные книгоиздательства уделяли очень мало внимания такого рода вопросам. Их обсуждение проводилось в основном в узких кругах: в отдельных проектных группах крупных компаний, на кафедрах университетов, в военных лабораториях. Со временем, обсуждение проблем разработки и внедрения стало производиться в периодической печати в виде статей и очерков. Одного из авторов наиболее известных очерков о проблемах разработки и внедрения программного обеспечения, доктора Фредерика Брукса (Frederick Brooks), без сомнения можно назвать современным философом.

Фредерик Брукс - профессор вычислительной техники в школе бизнеса Кенан университета штата Северная Каролина в Чэпел Хилл, США. Он известен, прежде всего, как "отец IBM System/360". Помимо этого, Брукс работал в IBM над архитектурой компьютеров Stretch и Harvest. Его сборник очерков "Мистический человеко-месяц или как создаются программные системы", впервые опубликованный в 1975 году, не утратил своей популярности и переиздается до сих пор. Причина его актуальности на протяжении более 30 лет бурного развития

технологий аппаратного и программного обеспечения заключается в том, что доктор Брукс освещает не только вопросы технологии. Главное в его работах - анализ самой природы программного обеспечения, его "сущности и акциденции", а также проблем, связанных с его разработкой и внедрением.

В работах Брукса разработка и внедрение больших программных систем сравнивается с доисторической смоляной ямой. Разработчики этих систем, подобно динозаврам, мамонтам и саблезубым тиграм, увязают в этой яме, пытаясь высвободиться из смолы. Но, чем отчаяннее борьба, тем сильнее затягивает смола, и как бы ни был силен или ловок зверь, в конечном итоге ему уготована гибель. Казалось бы, ничто в отдельности не вызывает трудностей - одну лапу всегда можно вытащить. Но накопление действующих одновременно и взаимовлияющих факторов все более и более замедляет движение.

И по сей день у компаний, разрабатывающих и внедряющих программное обеспечение, дела обстоят также. Такие смоляные ямы возникают постоянно во всем мире. Особенно эта ситуация характерна для разработки и внедрения ERP систем на отечественных предприятиях, в условиях "голодного" рынка, который прощает разработчикам и консультантам многие убийственные просчеты, которые вынужден оплачивать клиент.

Где же искать причину появления вязких смоляных ям и, главное, как увеличить производительность разработки и внедрения программного обеспечения, чтобы вырваться из них?

Во время выхода сборника в печать, большими системами для разработчиков являлись отдельные операционные системы и компиляторы для постоянно меняющихся аппаратно-программных архитектур. Именно такие системы обозначены Бруксом термином "Системный программный продукт". Сегодня сложность систем на порядки выше. В состав современных ERP систем входят целые комплексы серверов, рабочих станций, периферийных устройств, управляемых различными операционными системами. На них базируются распределенные базы данных и кросс - платформенные многоуровневые приложения, объединенные в локальные и глобальные вычислительные сети и сети передачи данных. Однако свойствами системного программного продукта обладают и современные системы, которые мы считаем большими сегодня.

Системный программный продукт

Широко известно заблуждение, что пара программистов в гараже может сделать замечательную программу, способную оставить позади разработки больших команд. И каждый программист охотно поверит в это, поскольку знает, что он лично способен писать программы со скоростью значительно превышающей среднюю производительность в больших промышленных командах.

Почему же до сих пор все профессиональные команды программистов не заменены одержимыми дуэтами из гаражей? Нужно посмотреть на то, ЧТО собственно производится!

Эволюция системного программного продукта представлена Бруксом в виде диаграммы (рис.1):Рисунок 1. Эволюция системного программного продукта

В левом верхнем углу диаграммы находится "Программа". Она является завершенным продуктом, пригодным для запуска своим автором на системе, на которой была разработана. В гаражах обычно производится именно такой продукт, и это - тот объект, посредством которого отдельный программист оценивает свою производительность.
Есть два способа, которыми "Программу" можно превратить в более полезный, но и более дорогой продукт. Эти два превращения обозначены на диаграмме стрелками. При перемещении вниз через горизонтальную границу "Программа" превращается в "Программный продукт". Это программа, которую любой человек может запускать, тестировать, исправлять и развивать. Она может использоваться в различных операционных средах и со многими наборами данных.
Чтобы стать общеупотребительным программным продуктом, программа должна быть написана в обобщенном стиле. В частности, диапазон и вид входных данных должны быть настолько обобщенными, насколько это допускается базовым алгоритмом. Примером может являться общепринятый стиль приложений MDI (multiple document interface), в котором выполнены программные пакеты Microsoft Office. Трудно себе представить современное бизнес-приложение, имеющее какой либо революционный интерфейс для общения с пользователем, не использующий стандартных окон и линеек прокрутки, предоставляемых операционными системами семейства Windows через интерфейс API (application programming interface), или не имитирующий его.
Затем программу нужно тщательно протестировать, чтобы быть уверенным в ее надежности. Для этого необходимо подготовить достаточное количество контрольных примеров для проверки диапазона допустимых значений входных данных и определения его границ, обработать эти примеры и зафиксировать результаты.
Наконец, развитие "Программы" в "Программный продукт" требует создания подробной документации, с помощью которой каждый мог бы использовать ее, делать исправления и расширять.
Доктор Брукс утверждает в своем очерке, что "Программный продукт" стоит, по меньшей мере, втрое дороже, чем просто отлаженная "Программа" с такой же функциональностью, что выражено на диаграмме знаком "x3" возле стрелки.
При пересечении вертикальной границы "Программа" становится компонентом "Программного комплекса". Последний представляет собой набор взаимодействующих программ, согласованных по функциям и форматам и вместе составляющих полное средство для решения больших задач. Чтобы стать частью программного комплекса, синтаксис и семантика ввода и вывода программы должны удовлетворять точно определенным интерфейсам.
Следовательно, программу нужно протестировать вместе с прочими системными компонентами во всех сочетаниях, которые могут встретиться. Это тестирование может оказаться большим по объему, поскольку количество тестируемых сочетаний растет экспоненциально. Оно также занимает много времени, так как скрытые

ошибки выявляются при неожиданных взаимодействиях отлаживаемых компонентов.
В правом нижнем углу диаграммы находится "Системный программный продукт". От "Программы" он отличается во всех перечисленных выше отношениях. И стоит, по мнению доктора Брукса, в девять раз дороже. Но это действительно полезный объект, который является целью большинства системных программных проектов, в том числе проектов по созданию и внедрению ERP систем. По моему мнению, в настоящее время индексы увеличения стоимости "Системного программного продукта" значительно выше, что только добавляет вязкости современным смоляным ямам.
Завершая обсуждение "Системного программного продукта", необходимо отметить его наиважнейшую характеристику - концептуальное единство (целостность). Концептуальная целостность архитектуры программного обеспечения, особенно если речь идет о больших системах, даже более важна, чем целостность архитектуры зданий. У большинства европейских средневековых соборов части, построенные разными поколениями строителей, имеют существенные различия в планировке и архитектурном стиле. Более поздние строители испытывали соблазн "улучшить" проект своих предшественников, чтобы отразить новые веяния моды и свои личные вкусы. Ярким, достойным восхищения, примером соблюдения архитектурного единства в противоположность распространенному смешению стилей является Реймский собор во Франции. Целостность архитектуры была достигнута благодаря самоотречению восьми поколений строителей собора, пожертвовавших своими идеями ради чистоты общего замысла.
Несмотря на то, что на создание программных систем не уходят века, в большинстве своем они демонстрируют меньшую согласованность архитектурных концепций, чем в любом соборе. Обычно это происходит не от того, что главные проектировщики сменяют друг друга, а потому, что проект расщепляется на задачи, выполняемые разными разработчиками.
В другом своем очерке "Серебряной пули нет - сущность и акциденция в программной инженерии" доктор Брукс рассматривает программное обеспечение (программный объект) как парную философскую категорию. Трудности, связанные с разработкой и внедрением программного обеспечения, по мнению Брукса, следует делить на:
· вытекающие из сущности программного объекта - внутренние, присущие природе программного обеспечения;
· вытекающие из акциденции программного объекта - сопутствуют разработке и внедрению программного обеспечения, но не являются внутренне ему присущими.
Сущность программного обеспечения
Сущностью программного объекта по Бруксу является конструкция, состоящая из сплетенных вместе концепций:
· наборов данных;
· взаимосвязей между элементами данных;
· алгоритмов;

· вызовов функций.
Эта сущность является абстрактной в том отношении, что концептуальная конструкция программного объекта остается одной и той же при различных реализациях. Тем не менее, она обладает высокой точностью и большим числом деталей.
Доктор Брукс считает, что "… сложность создания программного обеспечения заключается в задании технических требований, проектировании и проверке этой концептуальной конструкции, а не в затратах, связанных с ее представлением и проверкой точности представления…".
программное обеспечение методология внедрение
Трудно не согласиться с этим наиважнейшим тезисом Брукса: главные трудности разработки программного обеспечения начинаются задолго до начала программирования и могут существовать отдельно от него. Синтаксические или логические ошибки программиста случаются очень часто и сегодня, но они легко находятся и исправляются с помощью современных интегрированных сред разработки. Намного печальнее последствия концептуальных ошибок проектирования сложных программных систем. Поэтому возможность увеличения производительности разработки и внедрения программного обеспечения следует искать в его сущности.
Брукс выделяет следующие неотъемлемые свойства сущности программных объектов: сложность, согласованность, изменяемость и незримость. Остановимся подробнее на каждом из этих свойств.
Сложность
Сложность программных объектов более зависит от их размеров, чем сложность любых других создаваемых человеком конструкций, поскольку никакие две их части не схожи между собой (по крайней мере, выше уровня операторов). Если они схожи, то объединяются, как правило, в одну подпрограмму, функцию, класс. В этом отношении программные системы имеют глубокое отличие от компьютеров, домов и автомобилей, где повторяющиеся элементы имеются в изобилии.
Сами компьютеры сложнее, чем большинство изготавливаемых людьми вещей. Число их состояний очень велико, поэтому их трудно понимать, описывать и тестировать. У программных систем число возможных состояний на порядки величин превышает число состояний компьютеров.
Масштабирование программного объекта - это не просто увеличение в размере тех же самых элементов. Это обязательно увеличение числа различных элементов. В большинстве случаев эти элементы взаимодействуют между собой неким нелинейным образом, и сложность целого растет значительно быстрее, чем линейно.
Сложность программ является существенным, а не второстепенным свойством. Поэтому описания программных объектов, абстрагирующиеся от их сложности, часто абстрагируются и от их сущности. Математика и естественные науки достигли больших успехов, создавая упрощенные модели сложных природных явлений, получая из этих моделей свойства, и проверяя их опытным путем. Это удавалось благодаря тому, что сложности, игнорировавшиеся в моделях, не были

существенными свойствами явлений. Но это не работает, когда сложности являются сущностью.

Многие классические трудности разработки программного обеспечения проистекают из этой сложности сущности и ее нелинейного роста при увеличении размера. Сложность является причиной трудности процесса общения между участниками команды разработчиков, что ведет к ошибкам в продукте, превышению стоимости разработки и внедрения, затягиванию выполнения графиков работ. Сложность является причиной трудности перечисления и понимания всех возможных состояний программы, а отсюда возникает ее ненадежность. Сложность функций является причиной трудностей при их вызове, из-за чего программами трудно пользоваться. Сложность структуры является причиной трудностей при развитии программ и добавлении новых функций так, чтобы не возникали побочные эффекты. Сложность структуры является источником скрытых состояний, в которых нарушается система защиты.

Сложность является причиной не только технических, но и административных проблем. Из-за сложности трудно осуществлять надзор, в результате чего страдает концептуальная целостность системы. Обучение и понимание становятся колоссальной нагрузкой, из-за чего замены в команде разработчиков превращаются в катастрофу.

Согласованность

Люди, связанные с разработкой и внедрением программного обеспечения не одиноки в проблемах сложности. Физика имеет дело с объектами чрезвычайной сложности даже на уровне элементарных частиц. Но физик работает в твердой уверенности, что всему есть рациональное объяснение. Альберт Эйнштейн утверждал, что "… природа должна иметь простые объяснения, поскольку Богу не свойственны капризность и произвол …".

У разработчика программного обеспечения нет такой утешительной уверенности. Сложность, с которой он должен справиться, по большей части является произвольной, необоснованно вызванной многочисленными человеческими требованиями и системами, которым должны удовлетворять его интерфейсы. Системы различаются интерфейсами и меняются во времени не в силу необходимости, а лишь потому, что были созданы разными людьми.

Во многих случаях программное обеспечение должно согласовываться, поскольку только что появилось среди других систем. В других случаях оно должно согласовываться потому, что его легче согласовать. Но во всех случаях значительная часть сложности происходит от согласования с другими интерфейсами, и это невозможно упростить только в результате перепроектирования программного обеспечения.

Изменяемость

Программные объекты постоянно подвержены изменениям. Конечно, это относится и к зданиям, автомобилям, компьютерам и т.п. Но произведенные материальные вещи редко подвергаются изменениям после изготовления. Их заменяют новые модели или существенные изменения включаются в более поздние серийные

экземпляры той же модели. Но, то и другое случается значительно реже, чем модификация работающего программного обеспечения.
Отчасти это происходит потому, что программное обеспечение какой либо системы (бизнеса) воплощает ее процессы (бизнес процессы), а процессы более всего ощущают влияние изменений в предметной области системы (бизнеса).
Программное обеспечение легче (на первый взгляд) изменить: это чистая мысль, бесконечно податливая. Здания тоже перестраивают, но признаваемая всеми (сразу) высокая стоимость изменений усмиряет пыл новаторов.
Все удачные программные продукты подвергаются изменениям. При этом действуют два процесса:
· Как только обнаруживается польза программного продукта, начинаются попытки применения его на грани или за пределами первоначальной области. Требование расширения функций исходит, в основном, от пользователей, которые удовлетворены основным назначением и изобретают для него новые применения;
· Удачный программный продукт живет дольше обычного срока аппаратно-программной среды, для которой он первоначально был создан. Программа должна быть согласована с требованиями новой среды.
Таким образом, программный продукт встроен в культурную матрицу приложений, пользователей, бизнес правил и аппаратного обеспечения. Все они непрерывно меняются и их изменения неизбежно требуют изменения программного продукта.
Незримость
Программный продукт нематериален. А геометрические абстракции являются мощным инструментом. План здания помогает архитектору и заказчику оценить пространство, возможности перемещения, виды. Становятся очевидными противоречия, можно заметить упущения. Масштабные чертежи механических деталей и объемные модели молекул, являясь абстракциями, служат той же цели. Геометрическая реальность изображается в геометрической абстракции.
Реальность программного обеспечения не встраивается естественным образом в пространство. Поэтому у него нет готового геометрического представления подобно тому, как местность представляется картой, механизмы - чертежами, а компьютерные сети - схемами соединений. Как только мы пытаемся графически представить структуру программы, мы обнаруживаем, что требуется не один, а несколько графов, наложенных один на другой. Несколько графов могут представлять управляющие потоки, потоки данных, схемы зависимостей, временных последовательностей, соотношений пространств имен. Обычно они не являются иерархическими и даже плоскими. На практике одним из способов установления концептуального контроля над такой структурой является обрезание связей до тех пор, пока один или несколько графов не станут иерархическими.
Несмотря на прогресс, достигнутый в ограничении и упрощении структур программного обеспечения, они остаются невизуализируемыми по своей природе, тем самым лишая нас одного из наиболее мощных инструментов оперирования концепциями. Этот недостаток не только затрудняет индивидуальный процесс проектирования, но и серьезно затрудняет общение между разработчиками.

Итак, мы рассмотрели основные присущие программному обеспечению свойства, то есть определили, ЧТО должно создаваться в результате разработки и внедрения программного обеспечения вообще и ERP систем в частности. Теперь рассмотрим, КАК это следует делать.

Как создавать большие программные системы в разумные сроки?

Программные проекты чаще проваливаются из-за нехватки календарного времени, чем по всем остальным причинам вместе взятым. Почему эта причина неудач настолько распространена?

Во-первых, слабо развиты и редко применяются методы оценок. В сущности, они отражают молчаливое и совершенно неверное предположение, что все будет идти хорошо. Пессимистические оценки менеджерами команд разработчиков делаются редко.

Во-вторых, применяемые методы оценки ошибочно путают достигнутый прогресс с затраченными усилиями, неявно допуская, что скорость выполнения проекта пропорциональна количеству занятых в нем сотрудников.

В-третьих, поскольку менеджеры программных проектов не уверены в своих оценках, им часто недостает вежливого упрямства для убеждения клиента в том, что определенный программный продукт не может быть разработан и внедрен в более короткие сроки без существенной потери качества.

В-четвертых, при обнаружении отставания от графика естественной и общепринятой реакцией является увеличение числа разработчиков. Это все равно, что тушить пламя бензином. В результате дела идут значительно хуже. Чем сильнее пламя, тем больше нужно бензина, и в итоге этот путь приводит к катастрофе.

Рассмотрим подробнее перечисленные аспекты данной проблемы:

Во-первых, Оптимизм

Итак, первый краеугольный камень проблемы - излишний оптимизм. Все разработчики программного обеспечения - оптимисты. Сколько раз в течение часа можно услышать от одного и того же программиста: "На этот раз она точно пойдет!" или "Я только что выявил последнюю ошибку!".

В основе планирования процесса разработки и внедрения программ часто лежит ложное допущение, что все будет хорошо, то есть каждая задача займет столько времени, сколько "должна" занять.

Глубокий оптимизм разработчиков заслуживает отдельного изучения. Дороти Сэйерс (Dorothy Sayers) в своей книге "Разум творца" ("The Mind of the Maker") выделяет в творческой деятельности три стадии:

1. Замысел;
2. Реализация;
3. Взаимодействие.

Книга, компьютер или программа сначала возникают как идеальное построение, существующее не во времени и пространстве, а лишь в форме замысла в мозгу своего создателя. Реализация же во времени и пространстве происходит с помощью пера, чернил, бумаги, печатной машинки, устройств ввода информации в компьютер и т.п. Творение будет завершено, когда кто-либо прочтет книгу, воспользуется

компьютером или запустит программу, тем самым вступив во взаимодействие с разумом их создателя.
Для человека, который что-то создает, неполнота и противоречивость идей выявляются только при их реализации. Поэтому для теоретика изложение на бумаге, экспериментирование, изготовление являются неотъемлемыми частями творческого процесса.
Во многих видах творческой деятельности материал с трудом поддается обработке. Дерево колется, краски пачкаются, глина трескается, и т.д. Эти физические ограничения сужают круг идей, которые могут быть выражены, а также создают неожиданные трудности при реализации. И эти ограничения известны заранее. Реализация, таким образом, требует сил и времени как из-за физического материала, так и ввиду неадекватности основополагающих идей. Большую часть затруднений при реализации человеку свойственно объяснять недостатками физического материала, поскольку он "чужд" ему - в отличие от идей, которыми он гордится.
При создании же программ, мы имеем дело с чрезмерно податливым материалом. Программист осуществляет свои построения на основе чистого мышления - понятий и очень гибких их представлений. Поскольку материал столь податлив, разработчик не ожидает трудностей при реализации, отсюда и глубокий оптимизм. Из-за беспечной ошибочности идей разработчиков возникают ошибки в программах.
Для отдельной задачи допущение, что все будет хорошо, оказывает на график работ вероятностный эффект. Все может действительно идти по плану, поскольку есть некоторое распределение вероятности для возможной задержки и существует конечная вероятность того, что задержки не будет. Однако большой программный проект состоит из множества задач, часть из которых может быть начата только после окончания других. Вероятность того, что все задачи будут завершены в срок, для больших проектов бесконечно мала.
Следовательно, оптимизм разработчиков необоснован и не имеет оправдания.
Во-вторых, Человеко-месяц
Вторая ошибка рассуждений большинства разработчиков заключена в самой единице измерения, используемой при оценивании и планировании: человеко-месяц. Стоимость проекта действительно может быть измерена как сумма произведений месячных затрат на каждого занятого в разработке и внедрении на количество затраченных месяцев. Но не достигнутый результат! Поэтому:
"… использование человеко-месяца как единицы измерения объема работы является опасным заблуждением …" (Ф. Брукс)
Число занятых и число месяцев являются взаимовлияющими лишь тогда, когда задачу можно распределить среди работников, которые не имеют между собой взаимосвязей (рис.2):Рисунок 2. Зависимость времени от числа занятых - полностью разделимая задача

Если задачу нельзя разбить на независимые части, поскольку существуют ограничения на последовательность выполнения этапов, то увеличение затрат не оказывает влияния на исполнение графика (рис.3):Рисунок 3. Зависимость времени

от числа занятых - неразделимая задача

Многие задачи разработки и внедрения программного обеспечения относятся именно к этому типу, поскольку отладка, например, по своей сути носит последовательный характер.
Для задач, которые могут быть разбиты на части, но требуют обмена данными между подзадачами, затраты на обмен данными должны быть добавлены к общему объему необходимых работ. Поэтому наилучший достижимый результат оказывается несколько хуже, чем простое соответствие числа занятых и количества месяцев (рис.4):Рисунок 4. Зависимость времени от числа занятых - разделимая задача, требующая обмена данными

Дополнительная нагрузка состоит из двух частей:
· обучение;
· обмен данными.
Каждого работника нужно обучить технологии, целям и границам проекта, общей стратегии и плану работы. Это обучение нельзя разбить на части, поэтому данная составляющая затрат изменяется линейно в зависимости от числа занятых.
Намного хуже дело обстоит с обменом данными. Для трех работников требуется втрое больше попарного общения, чем для двух, для четырех - вшестеро (таб.1):
Таблица 1. Зависимость количества каналов попарного общения от количества занятых

Количество занятых

Количество каналов попарного общения

2

1

3

3

4

6

5

10

6

15

7

21

8

28

9

36

10

45

11

55

К одиннадцати занятым, количество каналов попарного общения пятикратно превышает количество занятых! И далее число каналов растет нелинейно. Дополнительные затраты на обмен данными могут полностью обесценить результат дробления исходной задачи и привести к положению, изображенному на рисунке

5:Рисунок 5. Зависимость времени от числа занятых - задача со сложными взаимосвязями

Поскольку создание системного программного продукта является по сути сложным проектом с множеством взаимосвязанных задач, временные затраты на обмен данными велики и быстро начинают преобладать над сокращением сроков, достигаемым в результате разбиения задачи на более мелкие подзадачи. В этом случае привлечение дополнительных работников не сокращает, а удлиняет исполнение графика работ.

В-третьих, Робость в оценках

Для разработчика программного обеспечения, как и для повара, давление со стороны клиента может определять запланированный срок завершения задачи, но не может определять время ее фактического завершения. Омлет, обещанный через две минуты, может успешно жариться, но если через две минуты он не готов, то у клиента есть две возможности:

· ждать еще;

· съесть омлет сырым.

Тот же выбор стоит и перед заказчиком программного обеспечения.

У повара есть еще одна возможность - добавить жару. В результате омлет, скорее всего, окажется безнадежно испорченным: горелым с одной стороны и сырым - с другой.

К сожалению, такие далекие от реальности графики, нацеленные на желаемую клиентом дату, встречаются в проектах по разработке и внедрению программного обеспечения довольно часто. Особенно в нашей стране, это приобрело характер эпидемии. Поэтому данный аспект проблемы особо актуален для отечественного рынка программного обеспечения вообще, и ERP систем в частности.

Суть данного аспекта проблемы заключается в том, что очень тяжело, рискуя потерять контракт, с энергией и любезностью отстаивать перед клиентом срок, который определен без применения каких-либо количественных методов при недостатке данных и подкреплен, в основном, только интуицией менеджера проекта.

В-четвертых, Тушим пламя бензином

Что делают, когда важный программный проект начинает отставать от графика? Естественно, добавляют людей. Как рассмотрено ранее и отображено на рисунках 2 - 5, это не всегда помогает. В случае разработки и внедрения больших ERP систем, ситуация в большинстве случаев складывается, как на рисунке 5.

Джерри Огдин в своей статье "Монгольские орды против суперпрограммиста" приводит множество подробных расчетов на конкретных примерах, в которых показывает, как добавление новых сотрудников к командам, отстающим от проектного графика, приводит в скором времени к растущей, как снежный ком потребности в новых сотрудниках. Именно эту ситуацию доктор Брукс называет "тушить пламя бензином".

Подводя итоги анализа основных факторов наиболее частой причины провала программных проектов, Фредерик Брукс формулирует свой Закон Брукса:

"Если проект не укладывается в сроки, то добавление рабочей силы задержит его еще больше".
Таким образом, Брукс в своей работе "Этот мифический "человеко-месяц", входящей в сборник его очерков, развенчивает миф о человеко-месяце. Доктор Брукс убедительно доказывает, что продолжительность осуществления программного проекта зависит от ограничений, накладываемых последовательностью работ. Максимальное количество разработчиков зависит от числа независимых задач. Эти две величины позволяют получить график работ, в котором будет меньше занятых разработчиков и больше месяцев. При этом возникает опасность устаревания разрабатываемого продукта, но нельзя составлять работающие графики, в которых занято больше людей и требуется меньше времени.
Итак, рассмотрев, чего нужно опасаться и чего делать не стоит при разработке и внедрении программного обеспечения, перейдем к поиску ответа на вопрос: "КАК следует это делать?".
Многие менеджеры, управляющие проектами, утверждают, что им предпочтительнее небольшие деятельные команды первоклассных специалистов, чем большие коллективы, состоящие из сотен программистов среднего уровня. И они правы, хотя бы в том, что само число разработчиков, действия которых нужно согласовывать, оказывает влияние на продолжительность и стоимость проекта, как было рассмотрено выше, поскольку значительная доля издержек вызвана необходимостью общения и устранения отрицательных последствий разобщенности, например системной отладкой.
Это также наводит на мысль, что желательно разрабатывать системы возможно меньшим числом людей. Действительно, зарубежный и отечественный опыт разработки и внедрения больших программных систем показывает, что подход с позиций грубой силы влечет удорожание, замедление, снижение эффективности, а создаваемые в результате системы не являются концептуально целостными. Общепризнано, что идеальная численность небольшой активной команды - 10 человек. Такая команда может быть эталоном управляемости и производительности. Но что делать, если требуется разработка и внедрение действительно большой системы, требующей временных затрат, скажем, в 5000 человеко-лет. Эффект, получаемый от снижения потерь производительности, растворится в таком большом проекте. Команде из 10 человек не завершить его в разумные сроки.
Исходя из этого, мы можем сформулировать главное противоречие процесса разработки и внедрения программного обеспечения: "маленькой команде потребуется слишком много времени для реализации действительно крупного проекта".
Разработчики больших программных систем постоянно сталкиваются с этой дилеммой. Для эффективности и концептуальной целостности предпочтительнее, чтобы проектирование и создание системы осуществили несколько светлых голов. Однако для больших систем необходим значительный контингент разработчиков, чтобы продукт мог увидеть свет вовремя. КАК можно примирить эти два желания? Ответ на этот вопрос тесно связан с третьим главным вопросом КТО? Кто та команда

разработчиков, которая сможет разрешить данное противоречие?

Операционная бригада

Известный американский ученый в области компьютерной техники и программного обеспечения, посвятивший множество работ методикам его разработки и внедрения, Харлан Миллз в 1971 году в своем докладе "Chief programmer teams, principles, and procedures" высказал революционную в то время идею, которая заключалась в следующем. Рассмотренное выше противоречие можно разрешить путем формирования достаточного для большой системы количества обособленных малых (около 10 человек) производительных и активных команд наподобие хирургической бригады. Общий проект должен разбиваться на крупные минимально связанные задачи, целиком поручаемые той или иной бригаде. При этом не каждый участник бригады будет врезаться в задачу, но резать будет один (Хирург), а остальные оказывать ему всевозможную поддержку, повышая его производительность и плодотворность. При такой организации немного голов заняты проектированием и разработкой, и в тоже время множество сотрудников работают "на подхвате". Будет ли такая организация работать? Кто играет роль анестезиологов и операционных сестер в группе разработчиков и как осуществляется разделение труда? Миллз делит свою бригаду на следующие роли:

Хирург

В докладе Миллза автор называет эту роль "Главный программист" (Chief programmer).

Он лично определяет технические условия на функциональность и эксплуатационные характеристики своей программы, проектирует ее, пишет код, отлаживает его и составляет документацию. Он пишет код на языке высокого уровня и имеет прямой доступ к компьютерной системе, на которой не только производится отладка, но и сохраняются различные версии его программ с возможностью легкой модификации файлов, а также осуществляется редактирование документации. Он должен обладать большим талантом, стажем работы свыше десяти лет и существенными знаниями в системных или прикладных областях.

Второй Хирург

Это второе "я" Хирурга.

Он может выполнять любую работу Хирурга, но менее опытен. Его главная задача - участвовать в проектировании, где он должен думать, анализировать, обсуждать и оценивать. Хирург испытывает на нем свои идеи, но не связан его предложениями. Часто Второй Хирург представляет свою бригаду при обсуждении с другими группами функций и интерфейсов. Он хорошо знает весь код программы Хирурга. Он исследует возможности альтернативных стратегий проектирования. Он подстраховывает на случай какой-либо беды с Хирургом. Он может даже заниматься написанием кода, но не несет ответственности за него в целом или за какую-либо его часть.

Администратор

Хирург - начальник, и ему принадлежит последнее слово в отношении персонала, прибавок заработной плате, помещений и т.п., но на эти дела он должен тратить как

можно меньше времени. Поэтому ему необходим профессиональный Администратор, заботой которого будут деньги, люди, помещения, машины и который будет контактировать с административным механизмом организации в целом. Возможно, на полный рабочий день Администратор должен привлекаться лишь в случае, когда отношения с заказчиком определяют существенные юридические, контрактные, отчетные или финансовые требования к проекту. В остальных случаях один Администратор может обслуживать две команды.

Редактор

Обязанность разработки документации лежит на Хирурге. Чтобы она была максимально понятна, он должен писать ее сам. Это относится к описаниям, предназначенным как для внешнего, так и для внутреннего использования. Задача редактора - взять созданный Хирургом черновик или запись под диктовку, критически переработать, снабдить ссылками и библиографией, проработать несколько версий и обеспечить публикацию.

Два секретаря

Администратору и Редактору нужны Секретари.

Секретарь Администратора обрабатывает переписку, связанную с проектом, а также документы, не относящиеся к продукту.

Секретарь Редактора обрабатывает документацию по разрабатываемому и внедряемому продукту.

Делопроизводитель

Он отвечает за регистрацию всех технических данных бригады в библиотеке программного продукта. Он имеет секретарскую подготовку и несет ответственность за все файлы, предназначенные как для обработки компьютером, так и для чтения участниками бригады.

Все данные для ввода в компьютер поступают делопроизводителю, который регистрирует их или вводит при необходимости с клавиатуры. Листинги вывода также поступают к нему для регистрации и хранения. Результаты самых свежих прогонов всех моделей заносятся в журнал результатов, а предыдущие хранятся в хронологическом порядке в архиве.

Особо важным в концепции Миллза является превращение программирования "из личного искусства в общественную деятельность" путем предоставления результатов всех прогонов всем членам команды и определения всех программ и данных как общей собственности команды, а не чьей-то личной.

Особые обязанности, возлагаемые на делопроизводителя, освобождают активных разработчиков от рутинных работ, систематизируют и обеспечивают надлежащее выполнение тех рутинных операций, которыми часто пренебрегают, и приближают главное, для чего работает команда - выпуск ее программного продукта.

В современной команде львиная доля этих обязанностей может быть выполнена с помощью CASE-средств. Наиболее яркий пример - системы управления изменениями.

Инструментальщик

В процессе разработки и внедрения программного обеспечения, особенно больших систем, команде постоянно требуются как аппаратные, так и программные ресурсы и

сервисы. И доступ к ним должен осуществляться с безусловной быстротой и надежностью. Только Хирург может решать, удовлетворяют ли его имеющиеся ресурсы и работа сервисов. Ему необходим Инструментальщик, ответственный за обеспечение доступа к ресурсам и сервисам, а также за создание, поддержку и обновление специальных инструментов. Сегодня это могут быть интегрированные среды быстрой разработки, средства коллективной работы, системы управления требованиями и изменениями, службы удаленного доступа к элементам процесса и т.д. У каждой команды должен быть свой инструментальщик, независимо от качества и надежности имеющихся централизованных служб. Он обязан обеспечивать необходимым или желаемым инструментом своего Хирурга, но не другие бригады. Инструментальщик занимает важное место в организации среды разработки и внедрения программного обеспечения.