МЕТОДОЛОГИИ

Повторное использование: готовые части ПО – ностальгия и дежавю

Идея о том, что программное обеспечение должно строиться из готовых «деталей», очень подробно рассмотрена в литературе по программированию (см. список лит-ры на стр. 92 [BELA80], [KERN76], [DENN811]).

Это чрезвычайно заманчивая идея. Во-первых, изготовитель ПО может сократить и расходы, и сроки, потому что уже имеются части программы, написанные заранее. Во-вторых, изготовитель ПО одновременно может повысить качество продукта, потому что заранее протестированная программа, как правило, более надежна, во всяком случае, она надежнее только что написанной. А поскольку расходы, сроки и качество – это показатели, которые часто вступают в противоречие друг с другом, найти методику, способную улучшить их все одновременно, особенно приятно. Таким образом, идея о готовых частях ПО выглядит поистине волшебно. В эпоху, когда слово «производительность» стало самым модным, готовые части ПО должны находиться в зоне повышенного интереса.

Здесь кроются два парадокса. Один, который обсуждается в работе [GLAS81], состоит в том, что применение готовых частей для создания ПО – это концепция восходящей разработки, и она, таким образом, вступает в противоречие с моделями нисходящей разработки 1970-х годов. Второй, более интересный, заключается в том, что принцип готовых компонентов составлял существо индустрии разработки ПО более четверти века тому назад. За прошедшие годы в сфере готовых компонентов наша отрасль претерпела значительный регресс. Очень важно ответить на вопрос «Почему?».

Пожалуй, примерно 95% тех, кто разрабатывает ПО сейчас, в 1950-х годах не имели к этой отрасли никакого отношения. Поэтому имеет смысл ненадолго вернуться в то далекое прошлое. В раскрывшейся перед нами панораме мы увидим признаки, предсказывающие наступление эпохи готовых компонентов ПО. В таком случае, конечно, у прошлого можно поучиться.

Шаг в прошлое

Зайдем со мной в вычислительный центр 1950-х. О, на двери надпись «Вычислительная лаборатория». Войдем внутрь.

Наше внимание тут привлекают несколько обстоятельств. Короткие стрижки программистов-мужчин, пышные прически программистов-

женщин. Шум клавишных перфораторов. Необъятные просторы машинного зала – всю эту площадь занимает компьютер, по сегодняшним меркам просто крошечный.

Все эти детали могут быть важными (или неважными) с точки зрения ностальгии, но с технической позиции они бессодержательны. Посмотрим повнимательнее. Туда, на рабочие столы программистов. Что это за руководство?

Берем его с книжной полки, смотрим: на переплете написано «SHARE» (Совместно используемое). Открываем его, читаем. Это каталог программных компонентов, и *у каждого программиста есть либо свой экземпляр, либо доступ к общему*.

«А это откуда?» – спрашиваем у ближайшего молодого программиста. (Интересно, что все они молоды – это нетрудно заметить. Совершенно новая область, как магнит, одним своим полюсом притягивает молодость... а другим отталкивает опыт!)

«А, это руководство SHARE, оно общее, – говорит он небрежно. – SHARE – это пользовательская группа нашего поставщика. Мы все пишем программы и добавляем их в SHARE, и мы все берем отсюда то, что написано другими».

«А что на этой странице? Это генератор псевдослучайных чисел с равномерным распределением. Откуда он?»

«Его написал Фред Маснер из United Technologies. Он вообще очень много написал для SHARE».

«А эту подпрограмму чтения символьной строки?»

«Билл Клинджер из Northwest Industries. Он пишет отличный код. Его программы всегда работают корректно».

Прервем ненадолго наш визит в 1950-е. Важно кое-что понять. Прежде всего, вычислительной техники как научной дисциплины, достойной упоминания, в 1950-е еще не было. До начала этой стадии развития оставалось еще почти десять лет. В программирование приходили математики, бизнес-администраторы и даже дипломированные специалисты по английскому языку! А это, в свою очередь, означало практически полное отсутствие литературы по вычислительной технике. Выходил журнал *Communications of the ACM*, но этого было слишком мало. И более широко распространенный *Datamation*. И недолго просуществовавший *Software Age*. Публикации в компьютерной периодике того времени были неважным способом повышения престижа индустрии ПО.

Было и еще одно обстоятельство. Стандартное ПО, которое продавцы поставляли вместе с компьютерами, еще не было разобрано на состав-

ляющие. На самом деле оно еще не было собрано! Компьютеры нередко продавали вообще без программ. Тут-то на сцене и появились пользовательские группы, подобные SHARE.

И именно это на самом деле придает смысл названию SHARE. Назначение этой группы было в том, чтобы делиться программным обеспечением, которое попросту больше негде было взять.

Второе приближение

Но вернемся в 1950-е. Теперь мы видим немного больше смысла в руководстве SHARE, представляющем собой коллекцию описаний готовых частей ПО. Посмотрим на него внимательнее.

Вот перед нами оглавление. Пробежав его быстро взглядом, мы видим, что части сгруппированы по функциональному назначению. Есть раздел с утилитами ввода/вывода, в другом разделе – работа с символьными строками, а вот группа математических функций... чего тут только нет. Откроем математический раздел, посмотрим, как он организован.

И в нем тоже ПО разбито на группы. Тригонометрические функции, подпрограммы для работы с матрицами и реализации численного интегрирования... есть и раздел с генераторами псевдослучайных чисел.

Ну что ж, углубимся еще немного. Что лежит в основе всей этой таксономии?

На этой странице нет ничего необычного. В первом абзаце описывается назначение компонента (функции, выполняемые программой). Далее мы видим имя автора и название фирмы, в которой он работает. Вот спецификация входных и выходных данных. И наконец описание ограничений и различные примечания. Мы видим, что одна часть обычно занимает одну страницу. Некоторые сложные программы, например утилиты ввода/вывода, занимают две-три страницы. Изредка, когда это важно, описывается алгоритм.

Но в любом случае в верхней части страницы приводятся имя автора и название фирмы. А внизу страницы, тоже всегда, отказ от гарантий. «Данное ПО протестировано, однако отсутствие ошибок в нем не гарантируется» или какая-нибудь аналогичная фраза.

«Программы-то хорошие?» – спрашиваем стоящего рядом программиста, озадаченные немного этими оговорками.

«Да почти всегда, – говорит он. – Почитайте код, и вы увидите, что обычно он очень хорош и – мне очень неприятно это признавать – лучше, чем самое гениальное, на что способен я. Как правило, в SHARE отдают только самое лучшее, потому что слишком многое при этом ставится на карту. Кроме того, если код плохой, это сразу видно».

«Слишком многое ставится на карту». «Если код плохой, это сразу видно». Все начинает представать в другом свете. Производство готовых частей ПО в 1950-е было дорогой к престижу и славе в индустрии ПО, а в современную эпоху стимул к тому, чтобы «опубликовать код или погибнуть», выражен слабо. Успех программиста, отдающего код в общее пользование, достигался очень большими усилиями. Некомпетентность отсеивалась автоматически.

Полистаем SHARE еще немного. Конечно, некоторые имена и названия встречаются на каждой пятой или десятой странице. Вот почему уже знакомый нам программист мгновенно вспомнил Фреда Маснера и Билла Клинджера, а также United Technologies и Northwest Industries. Это очень интересно. И даже потрясающе. Пожалуй, пора оставить 1950-е и осмыслить полученные сведения.

Что мы узнали

Давайте вернемся, пройдем через дверь с надписью «Вычислительная лаборатория» обратно в 1980-е. Что мы узнали?

Во-первых, технология производства частей ПО в то время существовала и процветала. Каждый программист мог рассчитывать на то, что при необходимости найдет готовую программу.

Во-вторых, были созданы классификация готовых программ и толковая поисковая документация. Было очень просто узнать, какие готовые компоненты ПО имеются в наличии.

В-третьих, статус автора программы был почетным. Компоненты поступали в общее распоряжение, потому что к тому имелась сильная мотивация. Люди, создававшие ПО, получали поощрения.

В-четвертых, не было душной атмосферы внутренних противоречий коллектива. Продавцы компьютеров не поставляли программное обеспечение бесплатно или по низкой цене – можно было либо делиться программами, либо писать их самостоятельно.

И в таком свете 1950-е – это забытое время программистов с короткими стрижками – являют собой замечательный образец для современной эпохи. Парадокс и ирония в том, что мы идем туда, где мы уже были. А теперь пора вернуться к вопросу «Что было сделано неправильно?».

Я видел, как это произошло. Это грустная и неприятная история.

Главная неправильность заключалась в самой иронии по поводу происшедшего. По мере того как 1950-е годы плавно переходили в 1960-е, постепенно начало выясняться, что создавать программное обеспечение становилось все труднее. Скажем, пакетами утилит ввода/вывода можно делиться, выкладывая их в руководство вроде SHARE, а вот мож-

но ли так поступить с операционной системой? Участники встреч группы SHARE (и других сообществ пользователей) все сильнее и сильнее давили на продавцов компьютеров, чтобы те включали ПО в поставки. И продавцы в конце концов так и сделали. Так идея совместного использования ПО почила... могли ли наши предки сделать это лучше и вернее? Встречи сообщества пользователей SHARE, открывающих свои решения для общего доступа, превратились в сборища пользователей, кричащих поставщикам «ДАЙ!».

Руководство SHARE вышло из употребления и в конце концов исчезло. Его место заняли километровые полки, занятые описаниями ПО от поставщиков компьютеров. В этих описаниях главное место отводилось системному ПО, много в них говорилось и об использовании различных инструментов внутри операционных систем, но понятие программных компонентов, за немногими исключениями вроде математических библиотек, попросту исчезло. В конце концов, изобилие таких компонентов заставило бы поставщиков намного активнее взаимодействовать с пользователями и сделало бы их более уязвимыми для критики. И разве могли бы они поместить отказ от гарантий под восхвалениями своих продуктов и уйти потом от юридической ответственности?

Произошло и еще кое-что, хотя роль этих событий в гибели сообщества пользователей программных компонентов была не так важна. Кафедры и факультеты вычислительной техники возникли в университетах по все стране, и постепенно стала появляться теория вычислительной техники. Энергия, которая тратилась на производство более качественных программных компонентов, теперь стала уходить на разработку более совершенных теорий их создания. Об этом лучше сказано в работе Белади (Belady) и Ливенворта (Leavenworth) [BELA80]:

> «...разработка ПО поляризована, и в ней существуют две субкультуры – созидателей и умозрительных мыслителей. Первые наделены способностями к эффективной разработке продукта, но не к экспериментированию и должны прибегать в своей деятельности к надежным методам, какими бы несовременными они ни были. Вторые изобретают, но не идут дальше того, чтобы сделать новинку достоянием гласности, и поэтому ничего не могут сказать о практической полезности (или бесполезности) идеи».

Мы все осознаем важность и приветствуем развитие теории программирования, но думаю, что совершенно забыли о значении роли творцов, созидателей программного обеспечения и даже не жалеем об этом. В действительности творец очень часто становится объектом публичной критики теоретиков [DENN81].

И последним событием было появление концепции «обезличенного программирования». Для разработки ПО, сложность которого постоянно

возрастала, требовалось все больше программистов, и поэтому материализовалась идея командной работы, витавшая в воздухе. И человеческое Я программиста, кажется, встало на пути успеха таких команд. Да, так оно и было. Командный подход все-таки не учитывал, что человеческое Я – это основной движущий механизм, который нельзя подавить, не получив отрицательных побочных эффектов. Можете ли вы представить себе обезличенного менеджера? Или обезличенного теоретика, публикующего в профессиональных журналах статьи без подписи и, следовательно, без отзывов? Наше эго дает нам мощную мотивацию, и если его убрать, результатом будет чувство летаргической безответственности.

Не морочьте себе голову

А теперь, чтобы замкнуть цепь этого рассуждения, зададимся вопросом: разве не именно это было той самой неправильностью, которая произошла с SHARE – старым, подлинно общим фондом ПО? В конце концов появился сильный авторитет (поставщик), который сказал, что возьмет на себя заботу о программных компонентах и инструментальных средствах и что программистам не надо забивать себе этим головы. А поскольку ничье эго не было озабочено тем, чтобы сделать вклад в общий фонд программного обеспечения, поступление программных компонентов в него прекратилось.

Так что можно сделать, чтобы ускорить наступление эпохи готовых частей ПО в современный нам период? Учиться на опыте 1950-х, конечно. В своем вычислительном центре:

1. Создать классификатор частей (компонентов) ПО и документацию к ним.
2. Предложить программистам пополнять этот сборник своими разработками.
3. Разработать и установить систему поощрений для тех, кто откликнется на этот призыв.
4. Распространить каталог компонентов среди всех программистов.
5. Решить, что надо сделать:
   a. сопровождать компоненты письменным отказом от ответственности, выбрав низкозатратный образ действия под девизом «Пользователь, берегись!»; или
   b. учредить организацию, которая будет заниматься централизованной сертификацией программных компонентов, что, конечно, повысит затраты, но позволит получить более надежный результат.

К чему приведет выбранный путь, постепенно прояснится. В пределах вашей организации – а может быть, и нескольких организаций – ра-

зовьется процветающая субкультура готовых частей ПО. На этой субкультуре вырастет коллекция компонентов ПО, разработанных специалистами прикладного программирования, то есть теми, которые, скорее всего, лучше остальных понимают, какие именно компоненты нужны. А система вознаграждений позволит выявить группу лучших программистов, обладающих «неповрежденным» эго, у которых появится новая причина гордиться тем, что они делают, и ощутимое материальное подтверждение этой причины.

Пятидесятые годы XX века, дежавю. Мы знаем, что это может произойти, потому что это уже было раньше. Именно так.

Список литературы

BELA80 – L. A. Belady and B. Leavenworth «Program Modifiability», IBM Re-search Report RC8147 (#35397), 3/6/80.

Описывает экспериментальный подход к определению значения абстракций данных для улучшения модифицируемости программ; в качестве тестовой модели выступала операционная система с объемом кода 100 тыс. строк.

DENN81 – P. J. Denning «Throwaway Programs», *Communications of the ACM*, February 1981.

Рекомендует создавать компоненты программ, пригодные для использования, как способ решения проблемы одноразового ПО.

GLAS81 – R. L. Glass «No One Really Believes in Top-Down Design». Software Soliloquies, *Computing Trends*, 1981.

Обращает внимание на те факты, что проектирование зачастую итеративно и что в проектировании всегда задействуется восходящая модель.

KERN76 – B. W. Kernighan and P. J. Plauger «Software Tools» Addison-Wesley, 1976.

Описание набора инструментальных средств программирования, подчеркивающее роль каждой из составных частей.

WEIN71 – G. M. Weinberg «The Psychology of Computer Programming», Van Nostrand Reinhold, 1971.

Здесь обсуждаются обезличенное программирование и вопросы прав собственности на программы.

Автоматическое программирование – слухи с вечеринки?

Не так уж часто в серьезной компьютерной литературе появляется что-то, что оказывает сильное влияние на образ действия и мыслей профессионалов, программирующих вычисления и обработку данных.

Первым, кого я помню, наверное, был Бен Шнейдерман (Ben Shneiderman) из Мэрилендского университета, который в 1977 году раскритиковал применение блок-схем алгоритмов, подкрепив критику практическими результатами. После публикации его работы сообщество программистов очень скоро отказалось от блок-схем алгоритмов как необходимой компоненты проектирования и документирования ПО, признав их фактически бесполезными.

А несколько позже Дэвид Парнас (David Parnas) и Фредерик Брукс (Fred Brooks), каждый в своей работе, написали, что прорывов в технологиях создания ПО, на которые многие возлагали надежды, не предвидится. Для ученых и практиков программирования это был холодный душ. Эти две статьи до сих пор продолжают оказывать свое воздействие.

За этими работами последовала еще одна. Чарльз Рич (Charles Rich) и Ричард С. Уотерс (Richard C. Waters), опубликовавшие статью в августовском выпуске *IEEE Computer Magazine* за 1988 год, обратили пристальное внимание на сферу автоматического программирования и приуменьшили связанные с нею ожидания до уровня, обозначенного ими как «слухи с вечеринки».

Слухи с вечеринки? Что они имеют в виду?

Многие годы практики, теоретики и менеджеры индустрии ПО с нетерпением ждали того времени, когда программы будут создаваться другими программами, а не программистами. Эта самая автоматизация программирования была бы похоронным звоном по программистам – если, конечно, мы правильно понимаем их истинные цели. Кое-кто даже объявил, что эта эпоха уже наступила, поскольку применяются языки четвертого поколения.

Рич и Уотерс, чья работа «Programmer's Apprentice»[1] освещала вопросы, самые актуальные в этой сфере, еще сильнее остудили головы многих своих коллег, одержимых идеей автоматического программирования.

---

[1] Rich, C., Waters, R. C. «The Programmer's Apprentice: a research overview» (Подготовка программиста: обзор исследований). *IEEE Computer*, Т. 21, №11, Ноябрь 1988, с. 10–25.

По Ричу и Уотерсу, в понятии автоматизированного программирования заключено несколько мифов. Вот они:

- Системы автоматического программирования, ориентированные на конечного пользователя, не требуют знаний в специальных областях. Неправда, говорят Рич и Уотерс. Если система автоматического программирования вообще должна существовать, то она должна быть экспертом не только по части генерирования программного кода, но и в конкретной прикладной области.
- Универсальные, полностью автоматизированные, предназначенные для конечного пользователя средства программирования возможны. И это неправда, говорят Рич и Уотерс. Все известные работы по автоматизированному программированию пренебрегают хотя бы одним из этих факторов.
- Требования к программе могут быть исчерпывающими. Еще один миф. Требования для задачи любой сложности представляют собой незаконченные аппроксимации, которые превращаются в окончательные решения в результате последовательных приближений.
- Программирование – это последовательный процесс. Миф! Программирование – это итеративный процесс, в ходе которого между программистом и конечным пользователем происходит непрерывный диалог.
- Программирования больше не будет. И это тоже миф! Конечные пользователи станут программистами, соединив свои знания в прикладных областях и новую компьютерную специализацию.
- Программирование как явление крупного масштаба прекратит свое существование. Это последний миф. Мы никогда не примиримся с тем, что будут решаться только задачи для узких областей и только немногими людьми.

Иначе говоря, процесс написания программ достаточно сложен, и некоторые исследователи попытались этот факт проигнорировать. Вклад Рича и Уотерса состоит в том, что они указали, чем нельзя пренебрегать. Видят ли Рич и Уотерс, если учесть вышесказанное, хоть какие-то перспективы для автоматического программирования? Да, но только, по их собственному выражению, «будущие системы автоматизированного программирования будут больше похожи на пылесосы, чем на самоочищающиеся печи». Автоматизация поможет программисту, но не сможет заменить его.

Авторы проводят интересное противопоставление между академическими исследованиями, очень амбициозными, но не слишком успешными, и коммерческими продуктами, создатели которых ставили и ре-

шали более скромные задачи. Известны коммерческие системы для следующих категорий:

- Системы запросов к базам данных, обеспечивающие извлечение информации из БД.
- Языки четвертого поколения (4GLs), которые делают возможным создание быстрых программных решений для узкоспециальных задач, притом что эффективность конечного программного продукта оставляет желать много лучшего.
- Генераторы программного кода, которые концептуально (по способу реализации) похожи на языки четвертого поколения (они работают, скорее, не как интерпретаторы, а как трансляторы в код на компилируемом языке).
- Высокоуровневые средства проектирования, которые представляют собой инструменты, помогающие проектировщикам.
- Инструменты управления проектами, помогающие менеджерам отслеживать выполнение задач.

Рич и Уотерс считают, что на пути к автоматическому программированию «достигнут значительный прогресс». Однако, подобно своим предшественникам Парнасу и Бруксу, они не видят, чтобы в этом направлении были сделаны какие-то прорывы.

Некоторые мысли о прототипировании

Индустрия ПО породила массу областей с интересными противоречиями. Прототипирование – одна из них.

Все началось с того, что некоторые крупные ученые и профессионалы сферы ПО занялись «жизненным циклом ПО», а именно последовательностью этапов, через которые, говорили они, проходит все программное обеспечение, развиваясь от состояния идеи, существующей в чьем-то мозгу, до законченного решения, функционирующего в компьютере.

Некоторые говорили, что жизненный цикл представляет собой удобный способ описания процесса разработки ПО. К сожалению, за эту идею ухватились некоторые формалисты и объявили, что отныне так *должно* разрабатываться все ПО без исключения. Короткое слово «должно» не слишком удлинило формулировку, но основательно изменило ее смысл.

Естественно, такое искажение идеи жизненного цикла встретило сопротивление. Дэн Маккракен (Dan McCracken), автор многих книг по языкам программирования, и Майкл Джексон (Michael Jackson), из-

вестный специалист по проектированию структур данных, объединились и написали статью протеста под названием «Life Cycle Concept Considered Harmful» (О вреде идеи жизненного цикла). В конце концов началась открытая полемика.

Тем временем некоторые практики и теоретики вели работы в другом направлении, отличном от идеи жизненного цикла. Их увлекла идея прототипирования, то есть построения одноразовой версии программного продукта, предназначенной для того, чтобы проверить идеи, лежащие в основе разработки, прежде чем зафиксировать их. Со временем оказалось, что прототипирование конфликтует с концепцией жизненного цикла, и потому критики жизненного цикла включили идеи прототипирования в свой арсенал.

Разработчики ПО, однако, продолжали исследовать и развивать идею прототипирования независимо от ее полемичности. Одно из самых интересных исследований было предпринято Полом Геккелем (Paul Heckel, работавшим над созданием ПО для переносного электронного словаря, устройства, которое умело бы переводить слова по отдельности (чтобы, например, помочь туристу при необходимости узнать, как по-французски будет «гамбургер»).

Никто до него не делал таких словарей, и задача Геккеля состояла не просто в том, чтобы написать программу, а в том, чтобы создать программное обеспечение, способствующее успеху продукта на рынке. Но благодаря чему не известный никому продукт может добиться успеха на рынке? Устройство должно иметь удобный размер, легко читаемый экран и дружественное ПО, которое помогает и которое любой новичок, не искушенный в электронных устройствах, мог бы применять, не впадая в ступор при каждой попытке что-нибудь перевести.

Оказалось, что этот проект как будто специально был придуман для прототипирования. Проблема, с которой столкнулся Геккель, коротко говоря, заключалась в том, что недостаточно четко были сформулированы исходные требования. Прототипирование помогло ему, позволив экспериментировать с различными вариантами продукта, который можно было подстроить для исследования меняющихся требований и найти такое их оптимальное сочетание, на котором можно было остановиться и создать программный продукт с четко определенными требованиями.

Конечно, прототипирование придумал не Геккель. Но он был одним из первопроходцев, и его статья «Designing Translator Software» (Проектирование ПО электронных словарей) в февральском выпуске *Datamation* 1980 года облекла всю концепцию прототипирования во вполне реальную и ясно различимую форму.

Остановимся и посмотрим на то, что сделал Геккель, и на противоречия между жизненным циклом и прототипированием. Вспомните, что при помощи прототипирования Геккель уточнял требования к продукту. Или, если смотреть на его работу с позиций жизненного цикла, он последовательно улучшал требования к продукту на соответствующем этапе жизненного цикла, пока не получал то, что его устраивало, после чего продолжал работу, переходя к проектированию и сборке окончательной версии продукта. Таким образом, жизненный цикл прототипирования, по Геккелю, выглядел так: требования – проектирование – реализация – фиксация требований и отбраковка продукта, а потом опять требования – проектирование – реализация.

И если вы считаете, что прототипирование и жизненный цикл противоречат друг с другу, то этот пример ясно показывает, в чем именно заключаются противоречия. Но работу Геккеля можно рассматривать как всего лишь небольшую вариацию на тему традиционного жизненного цикла. С той только разницей, что фаза определения требований у него была итеративной и включала проектирование и реализацию, предшествовавшие дальнейшему развитию остальной части жизненного цикла.

Однако такой взгляд на взаимодополняющую природу взаимосвязей между жизненным циклом и прототипированием не получил широкого распространения. Напротив, среди печатных работ преобладают посвященные теме «Прототипирование против точного описания...».

В середине 1984 года даже появилась пара статей, в которых различия между прототипированием и жизненным циклом исследовались экспериментально.

В настоящее время работы, посвященные экспериментальной оценке двух враждующих подходов, крайне немногочисленны, и мы должны приветствовать эти две статьи за предпринятую в них попытку рассмотреть противоречия объективно и бесстрастно. Статьи были интересными и хорошо написанными. Разве что противоречия были в них скорее обострены, чем приглушены.

В чем вообще была суть этих статей? В обеих документально фиксировался эксперимент, проведенный авторами и направленный на изучение преимуществ и недостатков двух «конкурирующих» методологий. Первая из этих статей называлась «Prototyping Vs. Specifying; A Multiproject Experiment» (Прототипирование или специфицирование; мультипроектный эксперимент) и была напечатана в журнале *IEEE Transactions on Software Engineering* в мае 1984 года. Исследование проводилось хорошо известным специалистом по разработке ПО Барри Боэмом (Barry Boehm) с коллегами.

Вторая статья, «An Assessment of the Prototyping Approach to Information Systems Development» (Оценка протопипированного подхода в разработке информационных систем), была опубликована на месяц позже в журнале *Communications of the ACM* и принадлежала перу Алави (Alavi) из Калифорнийского университета в Лос-Анджелесе (UCLA).

Базовая методика обоих экспериментов была одинаковой. Были задействованы несколько групп разработчиков, при этом некоторые создавали программный продукт, придерживаясь методологии жизненного цикла, а другие – прототипирования. По окончании разработки результаты оценивались в соответствии с хорошо продуманной системой критериев.

Что же показали результаты? Чтобы ответить на этот вопрос, были проведены интервью с пользователями и разработчиками ПО. Оказалось, что пользователи в целом оценивали продукты, созданные с применением прототипирования, как более удобные в работе, а отчеты, создаваемые этими продуктами, как более точные и полезные, однако (в эксперименте Боэма) функциональность ПО оставляла, по их мнению, желать лучшего.

Разработчики сообщали, что прототипированием труднее управлять, однако создаваемые проекты более логичны и последовательны, программный код легче писать и интегрировать. Таким образом, прототипирование в целом оказалось несколько более эффективным, но с определенными оговорками.

Так какое место занимает прототипирование? Центральное – в полемике. Известны интересные примеры его успешного применения, эксперименты, которые убедительно закрепляют этот успех, и некоторые важные концепции.

Как применять прототипирование на практике? Как оценить, сколько должен заплатить заказчик за программный продукт, созданный с применением прототипирования, если у вас нет четкого набора требований? Как избежать превращения первой версии продукта, сделанной наспех, в готовый продукт низкого качества?

На актуальность этих вопросов полемика не влияет. Уже больше десяти лет прошло с тех пор, как Фредерик Брукс сказал в своем знаменитом «Мифическом человеко-месяце», что первую версию продукта создают, чтобы выбросить ее, и мы до сих пор не знаем точно, как ответить на эти вопросы. Мы знаем только, что прототипирование – это перспективная технология, причем перспектива представляется вполне реальной.