**Материалы модуля «Алгоритмы ЧМВ»**

[Проект как прообраз системы 2](#_Toc219880005)

[Место пользователя в системе 3](#_Toc219880006)

[Принципы построения 4](#_Toc219880007)

[Область применения 7](#_Toc219880008)

[Процедура как суррогат поступка 8](#_Toc219880009)

[Легенда 9](#_Toc219880010)

[Принципы построения 10](#_Toc219880011)

[Принцип ограниченной осведомленности 10](#_Toc219880012)

[Принцип гарантированных навыков 10](#_Toc219880013)

[Принцип перекрытия процедур 10](#_Toc219880014)

[Принцип делегирования ответственности 11](#_Toc219880015)

[Область применения 12](#_Toc219880016)

[Информационные потоки и права доступа 13](#_Toc219880017)

[Вертикальные информационные потоки 14](#_Toc219880018)

[Модель секретности 14](#_Toc219880019)

[Модель надежности 15](#_Toc219880020)

[Горизонтальные информационные потоки 16](#_Toc219880021)

[Субъект-субъектная модель 18](#_Toc219880022)

[Субъект-объектная модель 18](#_Toc219880023)

### Проект как прообраз системы

Проективной мы будем называть человеко-машинную систему, в которой для взаимодействия с машиной человек составляет на языке **инструментальной** области проект, описывающий ее предполагаемое поведение. Типичная проективная система создается, например, на основе утилиты make. Для make объектами прикладной области являются файлы, а проектом - специальный файл по имени Makefile. В нем перечислены исходные объекты, целевые объекты (те, что нужны в результате), и правила изготовления целевых объектов из исходных: некоторые файлы получаются из других в результате компиляции или перекодировки, некоторые представляют собой архивы, некоторые могут просто копироваться, и т. д. По этому проекту утилита make строит дерево зависимости файлов друг от друга и выполняет указанные в проекте действия над исходными объектами, пока не получатся целевые. Если в процессе работы исходный объект изменился, при запуске make будут заново созданы только те промежуточные и целевые объекты, которые в конечном счете от него зависят. Утилита make придумана для сборки больших программных продуктов, но используют ее гораздо чаще - для автоматизации любых сложных действий над группами файлов (формирование документации, Web-страниц; планирование зависимых действий, в этом случае создаваемые файлы играют роль отметок о выполнении).

В проективной системе человек и машина работают над одной задачей, как правило, по очереди: сначала человек составляет проект (множество параметров, задающих структуру системы), потом машина выполняет его. Формально после изменения параметров система может перейти в состояние, совершенно не напоминающее предыдущее, поэтому говорят о сборке системы на основе проекта. Выполнение спроектированных действий может длиться сколь угодно долго (например, работа www-сервера), однако даже при наличии самых изощренных средств проверки правильности проекта невозможно в точности предсказать поведение системы на конкретном сложном примере. Поэтому в проективных системах сильно развиты **средства диагностики** состояния системы и качества готового продукта. Если пользователя что-то в наблюдаемом диагнозе не удовлетворяет, он исправляет проект и перезапускает систему. Так образуется цикл тестирования и отладки, характерный для взаимодействия пользователя и машины в проективной системе.

### Место пользователя в системе

Самый простой способ взаимодействия с проективной системой - метод проб и ошибок; это просто вырожденный вариант тестирования и отладки. Специфика этого метода применительно к проективной системе в том, что без знания устройства системы, без общего понимания того, как она начнет вести себя на основании сделанного проекта, на сто проб скорее всего придется сто ошибок, и эффективность метода будет нулевой. Посему главная часть проективной системы - полная и грамотная документация. Нет документации - нет системы. Вдумчивое чтение документации может свести количество проб к одной, а количество ошибок - к нулю (звучит невероятно, однако, если пользователь **достаточно** опытен, часто получается именно так).

Еще один простой способ взаимодействия с системой - создание проекта по примерам. Допустим, нам надо запустить почтовый сервер (сервер - это программа, а не компьютер!), да не просто так, а чтобы он принимал электронную почту только с определенных компьютеров сети. Выбираем для этого из примеров настроек (то есть проектов) почтового сервера тот, в котором заявлена "фильтрация почты". Дальше идет цикл тестирования-отладки. Запускаем сервер. Работает! В самом деле, откуда-то принимает почту, а откуда-то - нет. Заглядываем в проект. Там - порядка дюжины каких-то секций: одна отвечает за способ доставки почты, другая - за то, где и как ее хранить и т. д., все это мы узнаем, бегло глянув в документацию. Находим секцию, отвечающую за фильтрацию почты. Читаем соответствующий раздел документации повнимательнее. Оказывается, существует черный список отправителей и черный список компьютеров. Изменяем, перезапускаем сервер. Нет, нам нужен не черный, а белый список (перечень тех, от кого следует принимать почту). Читаем документацию еще внимательнее. Там есть и белый список, но в примере его нет. Исправляем настроечный файл сообразно документации, а черные списки удаляем и снова перезапускаем сервер. Если в остальном мы были внимательны, теперь он работает как надо.

Здесь показательны две вещи. Во-первых, сам пример нам не подошел, его пришлось исправить; причем исправить осознанно, на основе знаний о том, как работает система. Во-вторых, хотя проектирование по примерам особенно полезно на стадии обучения, специалист тоже частенько модифицирует старые проекты, чтобы не воссоздавать новые с нуля.

Часто встречаются простые задачи, для решения которых достаточно применить последовательно несколько системных утилит или составить небольшой тривиальный проект, каждая строка которого придает продукту требуемое свойство. Например, для того чтобы среди файлов каталога, содержащих слово "отчет", найти созданный позже всех, надо найти все файлы в каталоге, отобрать те, что содержат слово "отчет", отсортировать полученный список по времени создания файла и выбрать начало отсортированного списка (вот как это можно сделать из командного интерпретатора shell: ls -dt1 `grep -il отчет \*` | head -1). Назовем это прямым построением проекта. Прямое построение проекта возможно, только если свойства системы, описываемые проектом, строго соответствуют свойствам получаемого продукта. Фактически мы описываем именно свойства продукта, но на языке инструментальной области. Такие микрорешения микрозадач не нуждаются в проверке и их легко написать сразу, минуя цикл тестирования-отладки. К сожалению, далеко не все задачи решаются таким способом.

Самый распространенный способ работы в проективной системе - анализ поведения существующей версии системы и изменение проекта с учетом прогноза ее поведения в будущем. Это задача весьма непростая, тем более что добиться в результате надо не только безотказной работы системы, но и того, чтобы продукт получался качественный (имея в виду, что первое - необходимое условие второго). Назовем, по аналогии с математическим термином, этот вид взаимодействия человека и машины решением обратной задачи. По описанию того, как система на некоторых входных параметрах получает неудовлетворительный результат, требуется изменить входные параметры так, чтобы результат стал удовлетворительным. Не все обратные задачи можно решить; здесь нам помогает только знание структуры самой системы и понимание принципов ее работы. Более того, в процессе тестирования-отладки это знание как раз углубляется. Безусловно, столкнувшись со сложной задачей, смысл которой не исчерпывается перечислением свойств в проекте, надо тщательно изучать документацию по тем инструментам, которые предполагается использовать в решении.

Время, затрачиваемое на тестирование-отладку, будет тем меньше, чем лучше пользователь ориентируется в прикладной и инструментальной областях. На последней стадии владения проективным методом это время практически сводится на нет. Изучив задачу и пролистав руководство, сведущий пользователь почти безо всякой отладки создает сложный и довольно-таки неочевидный проект. Из чтения этого проекта совершенно не понятно, что задаваемая им система будет работать как надо, и тем не менее она работает как надо! Что важно: любую часть готового проекта автор в состоянии, если потребуется, обосновать. Из таких обоснований можно составить многочасовую лекцию, в то время как написание проекта от начала до конца заняло минут двадцать.

Однако не всегда сведущий пользователь столь добр. Как правило, он в ответ на такую просьбу иногда возмущается (значит, можно было и так догадаться), иногда говорит: "Делай так-то" (читай: вот решение, узнавай сам, почему оно таково) и только в сложных случаях объясняет. Эти отношения с системой именуют иногда satori. Вот так описывается "сатори", т. е. просветление, в философии Чань (Дзен):

"Признаки - нерациональность, постижение свойств изначальной природы, авторитетность без доказательств; позитивный результат - утверждение своего Я, экстаз, экзальтация, мгновенность, внезапность просветления; внешнее выражение - парадоксальные реакции: громоподобное молчание, оглушительные восклицания, безумный смех, сквернословие" [Из лекций проф. Г. Я. Стрельцовой, прочитанных на факультете психологии МГУ в 1996 г.].

Опытный пользователь решения свои строит на основе "выделения существенного", то есть тех аспектов задачи и тех параметров проекта, которые составят суть решения. Если не случилось стратегической ошибки, доделки будут чисто техническими. Так опытный шахматист за доли секунды способен определить выигрышность позиции, но не успевает запомнить ее целиком.

### Принципы построения

Принцип информационной открытости (И). Для существования и нормальной работы проективной системы очень важно, чтобы пользователь **мог узнать все** о работе любой ее части. Всякий инструмент должен быть документирован до степени, достаточной для понимания принципов его работы и всестороннего использования. В документацию нужно включать не только сведения о том, как этот инструмент применять, но и описание того, что именно и как именно он делает. Только такое знание позволит настроить его на решение конкретной задачи совместно с другими инструментами. Более того, если пользователь изготовил решение некоторого класса задач и не снабдил его полной документацией, это решение бесполезно для системы (пускай даже оно крайне полезно пользователю). Предпочтительнее менее мощный инструмент, но такой, которым в аналогичном случае сможет воспользоваться любой читатель документации. Более того, понимая устройство такого инструмента, другой пользователь сможет слегка улучшить его, приспособив к своему варианту задачи. Если инструмент удобен и понятен, так будут поступать многие, и довольно скоро он превзойдет по мощности и гибкости любой информационно закрытый программный продукт, для развития которого есть только один стимул: найти средства и заплатить разработчику.

Таким образом, информационно открытая система отлично приспособлена для разработки и эксплуатации "в сообществе" (community), то есть в ситуации, когда использовать, исправлять и развивать систему может, с формальной точки зрения, любой желающий, а на самом деле - именно тот, кому она нужна и кому это интересно (в случае информационной закрытости круг пользователей и разработчиков ограничивается "допущенными" до документации, причем, как правило, не разбираясь, интересно им это или нет). Еще одно достоинство информационно открытой системы заключается в том, что это единственная комфортная среда для любознательного человека. На любой вопрос он, если потребуется, может найти ответ. Такое положение дел позволяет непрерывно совершенствовать профессиональный уровень, оставляя себе наибольшую степень свободы. Любая задача представляется разрешимой, стоит только основательно изучить соответствующие инструменты (возможно, доработать их, или даже изобрести новые) и хорошенько подумать.

Принцип минимизации затрат (З). Из старой сказки: "Что быстрее всего? Мысль". Из новой: "Человек не должен работать. Работать должна машина, а человек - думать". Чего не умеет и никогда не сумеет машина? Мыслить. Что машина умеет лучше всего? Точно следовать указаниям. Поэтому в человеко-машинной системе мыслительные функции (вроде решения задач) стоит отдать человеку, а автоматические (вроде повторения действий) - машине. Более того, если человеко-машинная система вообще нуждается в человеке, то именно потому, что перед ней возникает известное число мыслительных задач, требующих решения. Ежели таковые имеются, все равно нужен кто-то, кто подтвердил бы их правильность. Значит, в хорошей проективной системе человек в основном должен **решать мыслительные задачи** (придумывать и составлять проект), а механических, бездумных действий совершать как можно меньше. Обдуманные действия сопровождаются, как правило, определенным внутренним усилием, которого требует принятие на себя ответственности за их результат.

Если, например, необходимо переименовать 20 файлов, то правильным решением будет составить одну команду - параметрический цикл, в котором новое имя каждого файла будет вычисляться из старого. Вводить же все 20 команд вручную не следует: нет гарантии, что где-нибудь на 17-й пользователь не ошибется при наборе. Чем меньше ручной работы, тем меньше вероятность механической ошибки. Взамен от человека требуется умение слегка программировать и некая решимость написанную микропрограмму счесть правильной, потому что цена ошибки возросла в 20 раз: если ошибиться, то будут неправильно переименовываться **все** файлы. При этом затраты (в основном умственные) на разработку решения могут быть весьма высоки (пришлось учиться программировать), затраты на реализацию должны быть сведены до минимума (цикл с одной параметризованной командой записывается явно короче 20 простых), а затрат на выполнение - никаких.

Заметим: затраты на **повторную** реализацию будут существенно меньше. Если будущая задача станет похожа на уже решенную, достаточно только подправить проект. Если это будет **та же самая задача**, главное - оформить решение так, чтобы потом вспомнить, что это именно оно, и чтобы им могли воспользоваться другие (см. И).

Принцип умопостижимости контекста (У). Для того чтобы задача могла быть решена, необходимо создать пользователю условия, в которых ее **удобно решать**. В частности, когда идет поиск решения и встает вопрос о выборе инструмента, очень важно отбросить как можно больше ненужных возможностей системы. Необходимо сократить контекст поиска до объема, который пользователь может целиком удерживать в голове. Доказано, что для этого в области выбора должно быть не более семи (максимум - девяти) однотипных частей (назовем это "правилом 7+-2", см. [[32]](http://www.excode.ru/popup.lit.html#32)). Поэтому инструментальная область должна поддаваться членению как раз на такие семь функционально различных частей, в каждой из которых тоже имеет смысл проводить семичастное деление, и т. д. вплоть до атомарных действий.

Учитывая, что уровней у такой пирамиды должно быть тоже не больше семи, получаем по меньшей мере 77=823 543 элемента в ее основании. Вполне достаточно... для идеальной системы. К сожалению, такой на свете нет. Однако и неидеальную систему следует разрабатывать с учетом ограниченных возможностей человеческой памяти и восприятия.

Другая сторона принципа умопостижимости контекста: даже самый сложный проект должен быть в целом понятен пользователю. Это можно назвать требованием human readable: любой проект должен быть таким, чтобы пользователь мог прочесть его и (после изучения документации) понять целиком. К примеру, программа, состоящая из двух функций по пять тысяч строк в каждой, представляет собой совершенно не читаемый проект (если только в ней не предусмотрен какой-то не относящийся к языку программирования способ разбиения на уровни).

Более строгое требование к проекту таково: достаточно опытный пользователь должен быть в силах не только просчитать, но и **написать** с нуля сложный проект (т. е. быть human writeable). Вообще-то с нуля никто сложные проекты не пишет, всегда найдутся какие-то предварительные заготовки, части старых решений и т. п. Но требование это гарантирует, что в проект не надо будет вставлять избыточную или не имеющую отношения к делу информацию. Так, модный нынче язык моделирования чего угодно XML не отвечает требованию human writeable, потому что содержит множество синтаксического шума: диаграмма из прямоугольника, эллипса и кривой Безье со стрелкой на конце в формате утилиты xfig занимает 356 байт и может считаться пригодной для чтения и записи вручную (правда, после преобразования некоторых числовых параметров в строковые), а вот аналогичная диаграмма, изготовленная при помощи утилиты dia, представляет собой 4,7 Кбайт на XML (стоит отметить, что ни тот, ни другой файл, строго говоря, не являются проектами, потому что и dia, и xfig суть визуальные средства разработки).

Принцип персональной ответственности (О). Уже не раз упоминалось, что пользователь проективной системы **берет на себя ответственность** за качество проекта, а стало быть, и за поведение системы, собранной на основе этого проекта, и за качество получаемого продукта. Деваться ему некуда: всякий раз, создавая решение даже самой немудреной задачи, перед тем как пустить его в ход, он должен сказать самому себе: "Да, это будет работать". То же самое он должен сказать своему начальнику и всем пользователям подвластной ему системы (если он системный администратор) или даже всей сети Internet. Самая элементарная команда работы с системой предполагает, что человек сначала подумал и принял решение. То, что команда может быть плодом невежества, безалаберности или глупости человека, в расчет не берется.

Столь трепетное и уважительное отношение к мнению человека выливается в достаточно суровое правило "захотел - получил": что бы ни творил пользователь, предполагается, что делает он это сознательно. Хочет удалить все свои файлы - что ж, значит, так надо (команда unix: rm -rf $HOME/\*. **ВНИМАНИЕ! Эта команда действительно сработает! Не повторять!** И кстати сказать, она удаляет **не все** файлы, см. главу 7). Восстановить удаленные файлы нельзя: вы же их сами удалили. Или надо было воспользоваться каким-нибудь другим, безопасным способом: вместо того, чтобы удалять ненужные файлы, переместить их в специальный каталог. Если пользователь нажал одну клавишу в текстовом редакторе - это команда редактирования, а не случайность, текст меняется. В редакторе, как и в любом инструменте разработки, конечно, есть функция отмены последнего действия: человеку свойственно ошибаться. Однако человеку свойственно и обдумывать решения, поэтому достаточно предусмотреть отмену **только последнего** действия. Правило "захотел - получил" здорово дисциплинирует, хотя на первых порах выглядит жестоко.

Второе правило: ответственность обусловлена **знаниями**. Хорошая проективная человеко-машинная система устроена так, что чем больше существует областей, в которых компетентен человек, тем больше он имеет возможностей изменять поведение системы, тем полнее управляет машиной. Пока вы не изучите основные принципы работы, скажем, процедур резервного копирования системных дисков, вы не сможете применять их. Или не станете, опасаясь взять на себя пока непосильную ответственность (за испорченную файловую систему). И наоборот, если человек знает, как улучшить работу системы, эти знания надо использовать. Если в придуманном решении есть доля творческого наития, вполне возможно, что никто другой **так** с этой задачей не справится.

Следствие 1. Очевидно, что основным направлением развития проективных систем будет создание все более мощных инструментариев, то есть наборов, позволяющих сравнительно быстро и эффективно строить решения задач в различных прикладных областях. Для программистов, например, это высокоуровневые библиотеки функций, специализированные модули языков программирования высокого уровня и сами эти языки. Разнообразие инструментов не нарушает принцип У: вряд ли программисту придется использовать их все **сразу**. Скорее всего, он, сообразуясь с профилем решаемых задач и стилем работы, подберет небольшой, но мощный подходящий инструментарий. Для разработки больших проектов, которые должны "уметь все", программист выберет высокоуровневую среду, где ему придется иметь дело с более сложными, но умопостижимыми объектами, при необходимости спускаясь на более низкий уровень. (Обычно в названии таких сред встречается слово toolkit - "инструментарий". Показательно, не правда ли?)

Следствие 2. Велосипедный парк. Некоторое количество (подчас изрядное) готовых решений придется собрать, что называется, **своими руками**, чтобы потом этими решениями пользоваться. И спустя какое-то время вы убедитесь, что занимались изобретением велосипеда - существуют инструменты более полные и, возможно, более удобные, чем ваш. Тут следует помнить четыре вещи. Во-первых, опыт и знание в проективной системе важнее конкретной поделки, так что ваши усилия не пропадут даром. Во-вторых, если не предвидится задач, которые ваша библиотека не решает, а чужая - решает, лучше оставить все как есть . В-третьих, когда чужая разработка объективно лучше, а задачи все прибывают, следует перейти на нее, если к тому нет формальных препятствий (лицензия, полная несовместимость и пр.). Лучше совместно совершенствовать мотоцикл, чем порознь пыхтеть над велосипедами. И в-четвертых, когда в следующий раз приметесь изобретать велосипед, оглядитесь вокруг в поисках готовых мотоциклов, то есть поищите подходящий для ваших задач инструментарий (например, на [www.sourceforge.net](http://www.excode.ru/www.sourceforge.net) или [www.freshmeat.org](http://www.excode.ru/www.freshmeat.org)).

### Область применения

Достоинства проективной системы - прямые следствия принципов ее организации. Многие из них уже упомянуты нами: **свобода ориентации** и возможность **совершенствоваться** - раз. Возможность, придумав и реализовав решение, оставить машину работать, а самому заняться новыми делами - два. Реакция проективной системы даже на самую нештатную ситуацию прогнозируема, потому что для прогноза нужно вдумчиво проанализировать проект, а он умопостижим (хотя предсказать, что в точности случится, нельзя). Сами нештатные ситуации **воспроизводимы**, так как состояние системы целиком описывается ее проектом плюс входным потоком данных, значит, легко обнаруживать узкие места и несообразности и устранять их. Проективные системы можно использовать для решения практически **любых** задач, дело только за временем, которое потребуется на осуществление решения.

Недостатки проективной системы тоже суть прямые следствия принципов ее реализации. Во-первых, много времени может потребоваться на ее **освоение**, причем чем больше человек делает в системе, тем выше должна быть его квалификация; а за обучение, как и за квалификацию, надо платить. И вот за свои деньги мы получаем человека, который много всего знает и много о себе думает, отказывается выполнять приказы, спорит, отсутствует на рабочем месте и т. п. - словом, ведет себя как творческая личность, а не как исполнительный сотрудник. Что поделать! Принцип О предполагает, что каждый делает свое дело с полной ответственностью и **принимает решения сам**. А если ответственности нет, творческий коллектив немедленно превращается в богему. Это во-вторых.

В-третьих, даже самые стандартные задачи проективная система выполняет всякий раз по-новому, потому что в ней заданы только параметры операций, а не сами операции (так, например, нельзя в точности предсказать, когда именно будет отослано из локальной очереди письмо, но это будет сделано обязательно). В-четвертых, количество циклов тестирование-отладка, которые придется пройти системе, прежде чем продукт ее будет признан качественным, **зависит от опыта** пользователя и строгости требований к продукту. В неудачных случаях задача так и остается нерешенной, гарантии решения нет никакой - кроме персональной гарантии самого пользователя и принципиальной разрешимости задачи.

У принципа информационной открытости **компьютерных** человеко-машинных систем есть еще одно немаловажное следствие. Основной инструмент такой системы - программа или библиотека. Самый надежный источник информации о нем - **исходный текст** на языке программирования. Более того, только если программа доступна пользователю в виде исходного текста, он может находить в ней ошибки, исправлять и развивать ее. Такую программу разрабатывает и отлаживает весь мир, точнее, все сообщество квалифицированных пользователей. Только доступ к исходному тексту программы гарантирует отсутствие в ней "вредоносных" частей (см. УК РФ, статья 273 и комментарии к ней).

Если исходные тексты программного продукта недоступны, это бьет сразу по всем принципам организации проективной системы. Во-первых, это нарушение принципа И. Во-вторых, это сильно ограничивает О, так как затрудняет (а чаще всего - запрещает!) **изменение** продукта. Даже обладая достаточными знаниями и будучи совершенно уверенным в своей правоте, человек не сможет решить задачу, связанную с доработкой инструмента. Для этого придется выдумывать дополнительное информационно открытое пространство **внутри** инструмента (например, дополнительно встроенный язык программирования). Мало того, что умножение сущностей противоречит У, само это синтетическое пространство будет основано на **легенде** об инструменте (см. лекцию 3), а не на действительной его структуре. При этом даже самое незначительное изменение свойств продукта может вылиться в дублирование этих свойств на "внутреннем языке", а тогда о З и говорить не приходится.

### Процедура как суррогат поступка

Процедурной мы будем называть человеко-машинную систему, доступную человеку в виде набора функций (процедур) внутри **прикладной** области, описываемых в терминах прикладной области и приводящих к наглядному или гарантированному изменению свойств объекта. Например, человеко-телевизор - полностью процедурная система: все задачи, которые ставит перед ней человек, описываются в терминах "программа", "громкость", "контраст" и т. п. Телевизор же (вернее, инструкция к нему) общается с пользователем на том же языке (кажется, в нем есть только один новый термин - "кнопка", все остальные, включая названия кнопок, повторяют известное пользователю).

Система в самом общем виде работает так: человек описывает, какой продукт он хочет получить, а машина рассказывает, какие действия надо для этого предпринять. Естественно, возможны варианты: часто человек не в состоянии вспомнить все свойства продукта, и машине приходится потихоньку выспрашивать их; так, например, устроен, "мастер подключения к Internet" в Windows 98 и прочие подобные ему "мастера" (wizards). Почти всегда стадия "какие действия предпринимать" этими мастерами и ограничивается: после того, как свойства продукта описаны, задачу уже можно решить, не посвящая пользователя в тонкости решения. Наконец (отличительная особенность процедурных систем!), под типовую пользовательскую задачу, скорее всего, найдется готовый вариант решения, так что пользователю останется только нажать кнопку, скажем, "форматировать текст" и наслаждаться результатом.

Существует некое предписание, или свод правил, согласно которому человек тянет за те или иные рычаги системы и получает на выходе определенные свойства продукта. Очень часто в роли предписания выступает инструкция по эксплуатации, или так называемый troubleshoot: вещи выходят недостиранными - кладите больше порошка, при работе слышен скрежет, а на вещах заметен известковый осадок - используйте средство против накипи, не гудит и не крутится - проверьте предохранитель, не поможет - включите вилку в другую розетку, если и это не поможет - звоните в службу ремонта. Предписания, составленные по принципу "совокупность признаков - действие", назовем табличными, потому что они обычно оформляются в виде двух колонок. Табличные предписания удобны, но практически всегда недостаточны. Если существует полностью задаваемая табличным предписанием человеко-машинная система, рано или поздно она становится автоматической, потому что выполнять процедуры из таблицы вполне под силу автомату, человек для этого не нужен. Системы кондиционирования, поддерживающие постоянную температуру и влажность в помещении, не нуждаются в человеке, который, глядя на градусник, дергал бы за ручку "холодно-жарко".

Поэтому в действительных системах предписание, скорее всего, будет неформальным. Неформальное предписание - это понятный пользователю текст, на основании которого он в некоторых ситуациях почувствует необходимость выполнить в системе определенные действия. Четкость, однозначность и полнота такого предписания, равно как и средства, вызывающие у пользователя желание действовать, остаются целиком на совести его составителей. Чаще всего в таких предписаниях ситуация описывается лишь приблизительно, сами рекомендации имеют вид доброго совета, а эффект применения слегка преувеличен. К примеру: "Если вам кажется, что с вашим диском что-то не в порядке, запустите программу проверки диска, которая быстро и эффективно устраняет все проблемы с диском". По сути дела, любое предписание, обещающее изменить не конкретное свойство объекта, а группу его свойств или объект в целом, неформально, так предполагает некий компромисс, допущение того, что действительный объект совпадает с заданным в предписании и что пользователю нужны именно те свойства, которые в результате у объекта появятся. К классу неформальных предписаний следует отнести и "интуитивно понятный интерфейс", раз уж создатели его уверяют, будто им можно пользоваться, не понимая до конца, что именно происходит.

### Легенда

Чтобы пользователь мог хотя бы отчасти планировать и прогнозировать поведение системы, он должен знать, как она устроена и как работает. А поскольку устроена система, как правило, непросто, приходится изобретать еще одну, упрощенную и не соответствующую действительности ее архитектуру, зато описанную в понятных пользователю терминах. Так возникает еще один способ взаимодействия с процедурной системой: посредством легенды. Легенда - это описание рычагов управления системой так, как если бы она только из них и состояла. Легенда хороша тем, что создает у пользователя ощущение, будто он отлично понимает устройство системы - пока следует предписанию. Недостаток легенды в том, что придерживаться ее всегда и во всем невозможно. Одни и те же - по легенде - объекты начинают вести себя в разных ситуациях по-разному, потому что соответствуют различным типам объектов системы. Типичный пример легенды - понятие иконки (значка, icon) в Windows. Иконки в обозревателе ведут себя не так, как некоторые иконки на рабочем столе, и совсем не так, как иконки в панели управления.

Пользователь процедурной системы зачастую не знает, как именно он добился от нее желаемого результата и далеко не всегда может с первого раза воспроизвести свои действия. Нажимал-нажимал на кнопки - и вышло. Как? А кто ж ее знает. Здесь работает накопленный опыт общения с системой, возможно, даже некое представление об ее истинном устройстве, добытое из системы в обход предусмотренных каналов информации и оттого не поддающееся формализации. Явление это имеет название black magic (черная магия) или voodoo.

### Принципы построения

#### Принцип ограниченной осведомленности

Меньше знаешь - крепче спишь. Устройство системы должно быть скрыто от пользователя, дабы не забивать его голову "ненужными" фактами и не давать ему возможности испортить систему. Кроме того, большую часть системы можно не документировать или же остановиться на стадии технической документации, продажей которой можно при случае и заработать. А самое главное, что недопущенный к тайному знанию пользователь за всяким исправлением системы придет к разработчику и опять-таки заплатит за upgrade. Пример: многие модели сотовых телефонов Siemens можно подключать к компьютеру, чтобы сохранять там телефонную книжку. Соединительный кабель, имеющий сертификат Siemens, стоит около 20 долл.. При наличии паяльника и схемы контактов можно было бы уложиться в 2-3 долл.. Поскольку техническая информация (схема контактов) восстановима из самого продукта (кабеля за 20 долл.), соединительный кабель для Siemens, изготовленный без сертификата (но технически эквивалентный) стоит порядка 5 долл. (материал+работа).

#### Принцип гарантированных навыков

Пользователь **не обязан** что-то особенное уметь для того, чтобы начать работать с процедурной системой, а если чему-то для этого все-таки надо учиться, то хотелось бы научиться как можно быстрее. Поэтому взаимодействие с процедурной системой, как правило, основано на уже имеющихся у пользователя или совсем несложных навыках. Нельзя основной упор делать на команды, требующие от пользователя решения мыслительной задачи одним сложным действием, лучше пускай задача решается механическим повторением ряда простых действий. Например, составлять команду "удалить текст от курсора до второго вхождения слова END" в этом смысле много хуже, чем, удерживая клавишу Shift, нажимать клавишу "стрелка вниз" до тех пор, пока образующееся выделение не достигнет строки, в которой слово END встречается второй раз. Потом, не отпуская Shift, следует подогнать курсор к этому слову, затем нажать клавишу Delete.

Но этот принцип можно понимать двояко. Ведь, с другой стороны, задачу надо решать прямо сейчас и времени на то, чтобы разбираться, как сделать это решение изящным и эффективным, нет. В конце концов, что важнее - процесс или результат? Очевидно, результат. Значит, в качестве инструмента решения требуется нечто такое, что гарантированно можно пустить в ход в любой ситуации, не расходуя время на изучение инструкций. Именно соблюдение принципа гарантированных навыков приводит к непременному возникновению Accessibility Tools: на самом-то деле не каждый человек может десять раз безошибочно нажать на стрелку, не отпуская при этом Shift, или десять раз подряд попасть указателем мыши по кнопкам очередного "мастера". С этим же связано обязательное присутствие в таких системах процедур Undo и Redo (причем чаще всего - многоуровневых), потому что механическая ошибка (и даже повторная!) - дело вполне обычное.

#### Принцип перекрытия процедур

Любая задача должна быть **хоть как-то решена**. Поскольку любые сложные действия со стороны пользователя приветствуются, только если их можно описать на языке прикладной области, средства интеграции (совместного применения) самих процедур в процедурных системах развиты слабо. Следовательно, неизбежна ситуация, когда один класс типовых пользовательских задач решается при помощи одной группы процедур (рычагов управления), другой класс задач - при помощи другой группы, но чем задача меньше похожа на типовую, тем сложнее придумать, какие процедуры и в какой последовательности применять, чтобы ее решить. Вывод: надо сами процедуры организовать так, чтобы с их помощью либо с помощью их суперпозиции (последовательного применения к одному и тому же объекту) можно было решить любую задачу. Это, конечно, невыполнимо, и стремление охватить предоставляемыми возможностями как можно большее пространство приводит к двум характерным свойствам системы.

Во-первых, процедур (кнопочек, рычагов, пунктов меню) в ней становится очень много. Чтобы пользователю легче было найти нужный рычаг, даже организуют специальные поисковые машинки. Иногда пункты меню располагаются так, что наглядность и простота не страдает (самые часто используемые - наверху, самые непопулярные - в глубине), но это приводит к тому, что самое интересное оказывается труднодоступным и про него никто и не вспоминает. Во-вторых, многие задачи так и приходится решать: путем последовательного применения нескольких других решений, используя при этом только часть их свойств (например, создать табличку, после чего вручную изменить толщину рамочек вокруг отдельных полей - это в случае, если ни один из готовых стилей таблиц не подходит). В самом деле, проще убедить пользователя в том, что готовое решение его устроит, чем подыскивать действительно удовлетворяющее его решение.

#### Принцип делегирования ответственности

Поскольку пользователь не имеет представления о том, что на самом деле происходит, когда он запускает очередную процедуру системы, **гарантию** качества выполняемых преобразований продукта может дать только **разработчик** процедуры. Пользователь может быть виноват лишь в том, что потянул не за тот рычаг, то есть на нем лежит ответственность за выбор процедуры. В еще большей степени от квалификации разработчика зависит качество получаемого продукта в целом: здесь разработчик отвечает и за непротиворечивость процедур системы, и за соответствие их работы стандарту, и за то, что все необходимые процедуры могут применяться именно тогда, когда они нужны, и за все остальное. Если процедуры слишком сложны, чтобы вдобавок отслеживать, не нарушают ли они какой-нибудь стандарт, проще вообще о нем забыть и придумать собственный. Таков, к примеру, формат файлов .doc: структура, правильность которой проверяется не соответствием какой бы то ни было документации, а пропусканием через процедурную систему, породившую их. Когда все же необходимо совершить какое-нибудь не вытекающее из предписаний действие, пользователь вынужден выполнять его на свой страх и риск. И уже никто ничего не гарантирует, потому что пользователю приходится по большей части руководствоваться тем, что раньше "в похожих случаях" система "так делала".

Мало того, пользователь процедурной системы в определенном смысле **привыкает к безответственности**. Всякое воздействие на систему должно обставляться - и обставляется - всевозможными предупреждениями наподобие "Вы уверены, что хотите сделать то-то и то-то? От этого все окошки могут позеленеть, а кнопочки - сморщиться!". Типичный признак процедурной системы - обилие запросов на **подтверждение** выбранного действия. Иногда подтверждения бывают двойные: за первым "Вы уверены?" следует "Вы действительно уверены?". Ответственный за систему - разработчик - **обязан** таким способом отмечать точки управления системой, в которых он перекладывает ответственность на пользователя. Выходит, пользователю предпочтительнее быть неуверенным - так меньше вероятность, что он что-нибудь испортит.

Следствие 1. Очевидно, что основным направлением развития процедурных систем будет создание все более полных и сбалансированных **наборов решений** для все большего числа прикладных областей. Не случайно слово "утилита" (utility) для обозначения программы пользователя было в таких системах заменено на "приложение" (application) (то есть те же рычажки, но в приложении к конкретной ситуации), а после - именно на "решение" (solution). Обычно, помимо средств решения основного класса задач, такие системы имеют некоторый запас возможностей "на все случаи жизни" (согласно принципу перекрытия процедур). Пользоваться таким урезанным запасом особенно неудобно. Зачастую, когда на одном компьютере соседствуют несколько процедурных систем, например CorelDraw, PhotoShop, MS Office и т. д. под одной Windows, их части вытесняют друг друга (вплоть до потери функциональности), напрасно "съедают" ресурсы компьютера и загромождают пространство.

Следствие 2. Диалог машины и пользователя в процедурной системе строится на **активности машины**. Это и всевозможные "мастера", и всплывающие подсказки, и венец творения - скрепка, что бьется головой об экран и провоцирует пользователя на всевозможные действия, о которых он без нее и не подумал бы. Дело в том, что процедурная организация человеко-машинной системы подавляет инициативу. Во-первых, согласно принципу делегирования ответственности, если в предписании сказано, что для получения нужных свойств объекта нужно выполнить 18 команд, то их и следует выполнять, а не те три, которые кажутся подходящими, но нигде не сказано, что так тоже можно. Вспомним и о неуверенности, в которой следует пребывать пользователю. Во-вторых, никто, за исключением, быть может, разработчиков, не знает, что на самом деле произойдет с объектом от воздействия этих трех команд. Согласно принципу ограниченной осведомленности, пользователю неизвестно ничего, кроме легенды, и ее стоит придерживаться в любом случае. В-третьих, система, состоящая почти сплошь из готовых решений, не предполагает, что перед ней будут ставить нерешенные вопросы.

### Область применения

Каковы достоинства процедурной системы? Во-первых, **быстрота** и **однозначность** реакции на предусмотренные ситуации. Если возникает задача, решение которой известно (а таких много), то дальнейшее - дело техники. Если задача нетиповая, то есть не имеет готового решения, ее либо можно разбить на последовательность типовых подзадач, либо скрепя сердце подогнать под существующую типовую и признать получившееся решение решением исходной задачи. Во-вторых, как только появляется возможность формализовать предписание, управляющее работой системы, человека можно бестрепетно заменять **автоматом** (как, например, автоматическая проверка диска в Windows). В-третьих, ускоренная программа обучения пользователя процедурной системы требует в основном заучивания и практически никаких знаний ни об устройстве машины, ни даже о самой системе. Наконец, ответственность за применение системы можно легко переложить на разработчика или на начальника, который приказал потянуть за рычаг; а это непременное требование в сферах деятельности, где высокая цена риска сочетается с информационными или должностными ограничениями.

Недостатки процедурной системы. Один из них уже упоминался в следствии 2: **подавление инициативы**. Кроме всего прочего, человеку может быть чрезвычайно некомфортно работать в условиях, когда по своей воле он ничего ровным счетом сделать не в состоянии. К тому же подмена знаний о внутреннем устройстве системы легендой влечет за собой **необратимость всякого воздействия**: даже если в наборе процедур к "прямому" изменению объекта предусмотрено "обратное" (к примеру, вставка и удаление), после их поочередного применения исходный объект уже не получится. Скорее всего, он будет нести в себе следы прежнего объекта, а также и "прямого", и "обратного" изменений. Тут-то и приходит на помощь кнопочка Undo, которая не оперирует обратным воздействием, а работает с архивом версий объекта, выбирая оттуда предыдущую его копию. Кроме того, как неизвестно, каким образом система работает, так неизвестно и то, каким образом она **не работает**, то есть отчего происходят ошибки, отчего именно это сочетание процедур приводит к сбою и т. п. Словом, невозможно предсказать, когда и как именно произойдет следующий сбой системы, а когда он произошел - понять, что его вызвало, и как с этим бороться. Наконец, чем дальше задача от типовой (для которой имеется готовое решение), тем сложнее ее такой системе решить и тем сложнее предсказать **качество продукта**, который получится в результате использования не вполне подходящих процедур.

Процедурные системы незаменимы там, где в основе взаимодействия человека с машиной лежит один из перечисленных выше принципов: если сложность устройства системы не соответствует выполняемым функциям (телевизор, стиральная машина), если необходима мгновенная отдача от неквалифицированного пользователя (системы продажи билетов, компьютерные игры), если оператор системы не может брать на себя ответственность за качество ее работы (рабочее место оператора банковской системы, консоль ракетной установки ПВО).

Идеальной была бы такая процедурная система, в которой процедуры перекрывались бы не более двух раз, интерфейс был бы нагляден, набор готовых решений охватывал бы все мыслимые задачи и не было бы понятно, как на самом деле все реализовано. Словом, система должна быть очень большой, у всех ее частей должен быть один разработчик и она сама должна "понимать", что человеку нужно, предлагая в разных ситуациях воспользоваться той или иной своей частью.

# Информационные потоки и права доступа

Объектом системы мы будем называть любой ее идентифицируемый ресурс (например, файл или устройство). Доступом к объекту системы - некоторую заданную в ней операцию над этим объектом (скажем, чтение или запись). Действительным субъектом системы назовем любую сущность, способную выполнять действия над объектами (**имеющую к ним доступ**). Действительному субъекту системы соответствует некоторая абстракция, на основании которой принимается решение о предоставлении доступа к объекту или об отказе в доступе. Такая абстракция называется номинальным субъектом. Например, действительным субъектом будет шпион, входящий в секретную лабораторию, а номинальным - украденная им пластиковая карта-пропуск (точнее, заложенный в нее код доступа).

Свойство операционной среды **разделять ресурсы** предусматривает ситуацию, когда некий субъект системы не может иметь доступ к определенному ее объекту. Основания для отказа в доступе могут быть разными. Например, двум пользователям нельзя одновременно записывать данные в один и тот же файл. Но еще чаще бывает, что пользователь вообще **не имеет права** записывать данные в этот файл - неважно, по распоряжению начальства или по соображениям системной безопасности. Сотрудник фирмы, которому не разрешено читать личные дела других сотрудников, не должен иметь доступа к ним и в электронном виде. Ограничения доступа субъектов (пользователей) к объектам (информации) должны строго соответствовать положению дел за пределами системы.

Свод правил, управляющих разграничением доступа, называется политикой безопасности системы. Идеальная политика безопасности должна быть **полной**, **непротиворечивой** и рассматривать **все** возможности доступа субъектов системы к ее объектам. Только соблюдение всех трех принципов гарантирует, что нарушить установленные правила (например, получить несанкционированный доступ к объекту) системными средствами невозможно. Если же предполагаемый злоумышленник воспользовался каким-нибудь **внесистемным** средством и смог получить статус номинального субъекта, к которому он не имеет отношения (например, подглядел чужой пароль и работает под чужим именем), никаких гарантий быть не может.

Полнота политики безопасности означает, что в ней должны быть отражены все существующие ограничения доступа. Непротиворечивость заключается в том, что решение об отказе или предоставлении доступа конкретного субъекта к конкретному объекту не должно зависеть от того, какими путями система к нему приходит. Третье требование, называемое также отсутствием недокументированных возможностей, должно гарантировать нам, что доступ не может быть осуществлен иначе как описанным в политике безопасности способом.

Из этого третьего требования сразу же следует, что в политике безопасности необходимо учитывать также и случаи **косвенного** доступа, например когда пользователю не разрешено открывать файл, но разрешено запускать программу, которая имеет право открывать этот файл. К тому же выясняется, что любая ошибка в **реализации** правил доступа (например, при написании программ) тоже может отразиться на безопасности системы.

### Вертикальные информационные потоки

Если данные в системе неравнозначны (например, есть секретная и несекретная информация или информация более и менее достоверная), придется определить еще и правила, помогающие соблюдать секретность и достоверность, т. е. отрабатывать политику безопасности. Как показано в [[31]](http://www.excode.ru/popup.lit.html#31), к самому понятию безопасности можно подходить по-разному. Остановимся на двух популярных моделях безопасности: так называемые "модели секретности" и "модели надежности". Эти две модели - часть практического результата теории информационных потоков, описывающего **вертикальные** информационные потоки.

Согласно общему предположению, данные в системе считаются **неравнозначными**: некоторая информация имеет высокую значимость, некоторая - низкую, возможно введение нескольких **уровней** значимости. **Потоком** информации при этом считается перемещение данных с уровня на уровень (понижение и повышение значимости), а политика безопасности регулирует такие потоки с целью сохранить некоторое основное свойство информации.

#### Модель секретности

В модели секретности таким свойством считается конфиденциальность информации. Модель повторяет общепринятое отношение к секретам: каждому объекту системы присваивается определенный уровень секретности, и чем выше уровень, тем меньше субъектов системы имеют к этому объекту доступ.

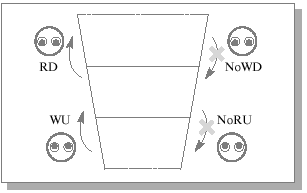
Обычно предполагается, что доступ организован **строго иерархически**: субъект имеет доступ к данным определенного уровня секретности **и ниже**. Для этого каждому **субъекту** присваивается единственный уровень доступа (для простоты будем заменять словосочетания "уровень секретности объекта" и "уровень доступа субъекта" на "уровень объекта" и "уровень субъекта"). Получившаяся схема напоминает пирамиду, откуда пользователь может "увидеть" только нижнюю ее часть, а все, что выше его уровня, становится "невидимым". Такое понимание секретности даже формально не может гарантировать неразглашение конфиденциальной информации.

В модели рассматривается два метода доступа к данным: чтение и запись. Если субъект перемещает некоторый объект системы на свой уровень - это операция чтения. Доступ по чтению не требует изменения уровня объекта, но предполагает, что в результате операции данные одного уровня будут доступны на другом. Принято считать, что конфиденциальность информации сохранится, если запретить доступ к данным более высокого уровня. Для чтения это верно.

Если субъект перемещает некоторый объект системы своего уровня на какой-нибудь другой - это операция записи. В результате операции записи может не произойти передачи данных, однако изменение уровня объекта (в обычном понимании - засекречивание или рассекречивание) позволит субъектам других уровней в дальнейшем иметь доступ к нему по чтению. Поскольку рассекречивание объекта нарушает конфиденциальность информации, эту операцию надо запретить, что означает запрет записывать данные на более низкий уровень секретности. Этот запрет (как и предыдущий) должен быть реализован на **системном уровне**, чтобы исключить саму возможность разглашения данных одними только системными средствами (без помощи взяток, шпионажа, шантажа и прочих внесистемных воздействий на субъекта).

Таким образом, правила нечтения верхнего уровня (No Read Up, NoRU) и незаписи на нижний уровень (No Write Down, NoWD) в модели секретности запрещают нисходящие информационные потоки. Заметим, что **запись на верхний уровень** (WU) правилами не запрещена: любой нижестоящий субъект вправе информировать вышестоящие инстанции о том, что происходит на его уровне (см. [рис. 9.1](http://www.excode.ru/art6280p1.html#image.9.1)).

**Рис. 9.1.** Вертикальные информационные потоки: модель секретности



Главный недостаток полностью реализованной модели секретности заключается в том, что вся информация - если она вообще куда-то движется - будет в конце концов засекречена. Некоторые субъекты совсем ничего не будут знать, прочие же станут "невыездными" и будут под страхом неминуемого наказания угрюмо молчать на вечеринках или в одиночку пить горькую по домам. Если такое положение дел не устраивает, есть три способа его преодоления.

Во-первых, можно ввести наивысший уровень секретности (с номером, допустим, -1), попадая на который объекты **становятся недоступны ни одному субъекту системы**. С ролью наивысшего уровня секретности хорошо справляются спецслужбы или машина для уничтожения бумаг. Если при этом в системе имеется приток объектов невысокого уровня секретности, можно добиться баланса.

Во-вторых, можно ввести автоматическую процедуру рассекречивания, основанную на свойствах объекта: его актуальности, времени жизни, связях с другими объектами и т. п. Говорят, что британские спецслужбы понижают секретность документов, если к ним не было обращений последние 50 лет.

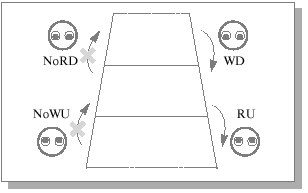
В-третьих, в системе можно предусмотреть доверенного субъекта - выделенного субъекта системы, которому **разрешено** нарушать политику безопасности. Он-то и будет рассекречивать некоторые документы, руководствуясь не системными, а личными правилами. Доверенных субъектов может быть несколько, каждый с правом нарушать только некоторые правила политики безопасности, они могут образовывать собственную иерархию и подчиняться собственной политике безопасности. Здесь возникает извечный вопрос: "Кто проверяет того, кто проверяет?", на который мы отвечать не беремся.

#### Модель надежности

Другой подход к пониманию безопасности описывает модель надежности. В этом подходе значимость объекта понимается как **степень доверия** к нему: например, может оцениваться качество содержащейся в нем информации, ее актуальность, достоверность и т. п. Надежность объекта сама собой повышаться не может, а может только падать: со временем информация теряет актуальность, да и степень доверия к ней не может быть выше степени доверия к ее источнику. Можно предположить, что эта модель будет зеркальна модели секретности, в которой значимость объекта, наоборот, никогда не понижается.

Как и в случае модели секретности к очевидному правилу, запрещающему запись на верхний уровень (субъект не в состоянии подтвердить, что его объект надежнее его самого), добавляется чуть менее очевидное, запрещающее чтение нижнего уровня (с какой это стати мы будем доверять объекту больше, чем какому бы то ни было породившему его субъекту?). Стало быть, налицо запрет на восходящие информационные потоки, который выражается в правилах NoWU и NoRD (см. [рис. 9.2](http://www.excode.ru/art6280p1.html#image.9.2)).

**Рис. 9.2.** Вертикальные информационные потоки: модель надежности



Естественно, в модели надежности возникает все та же ситуация "стекания" объектов на один уровень - на этот раз на самый нижний. Действительно, если нет повода повысить уровень доверия к объекту, а повод понизить этот уровень нет-нет да и появляется, то в конце концов все объекты вместо доверия начнут внушать одни подозрения.

Преодолеть вырождение уровней можно все теми же способами. Во-первых, можно организовать внесистемный приток и отток информации (т. е. пополнять систему новыми объектами достаточно высокого уровня и уничтожать потерявшие актуальность объекты при выбывании с нижнего уровня). Во-вторых, придумать алгоритм автоматического повышения уровня объекта (например, вести статистику успешного использования или что-нибудь подобное). В-третьих, привлечь эксперта (доверенного субъекта), который, исходя из внесистемных критериев, будет регулярно повышать уровень разных объектов системы (нарушая тем самым правило NoWU).

Рискнем предположить, что система, **полностью** обеспечивающая **оба** требования к информации (надежность и секретность), невозможна. Свод правил, учитывающий положение субъектов и объектов сразу в **двух** иерархиях, наверняка будет чрезвычайно сложным. Непонятно до конца, как обрабатывать ситуации вроде той, когда с точки зрения секретности один сотрудник имеет право передавать некий документ другому, а с точки зрения надежности - нет. Любой компромисс между двумя требованиями приведет к **нарушению** какого-нибудь из них. Поэтому **запрет** обычно считается важнее разрешения, но тогда ограничение **допустимых** в каждой модели потоков сильно снижает гибкость системы. Видимо, надежности и секретности одновременно можно достигнуть только несистемными средствами. Даже система, последовательно реализующая одну политику безопасности, для продолжительной работы требует некоторых внесистемных средств балансировки уровней.

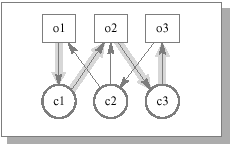
### Горизонтальные информационные потоки

Предположим теперь, что объекты в системе невозможно распределить по уровням значимости. Это бывает довольно часто, если значимость данных определяется не самой системой, а конкретным человеком, который с ними работает. Тогда для одного субъекта системы некоторый объект будет значим, а для другого - нет. Более того, иногда единственным поводом открыть или запретить кому-то доступ к объекту становится мнение некоторого субъекта системы.

Операции чтения и записи определяются в этом случае по отношению к каждому отдельному объекту (классу объектов). Чтение - это копирование данных из объекта в некоторое личное информационное пространство субъекта (пользователь **запоминает** содержимое файла). Запись - это копирование данных из личного информационного пространства субъекта в объект (пользователь изменяет содержимое файла). Если множество объектов, которые субъекту разрешено читать, не совпадают с множеством объектов, доступных ему для записи, а некоторые из записываемых объектов разрешено читать другому субъекту, возникает очевидный **горизонтальный** информационный поток. Точнее было бы назвать его безуровневым, однако слово "горизонтальный" здесь подчеркивает равную с точки зрения системы значимость и объектов, и субъектов.

Для выявления всех возможных информационных потоков в системе составляется диаграмма достижимости. Диаграмма достижимости состоит из вершин двух видов: субъектов и объектов, соединенных стрелками: от объекта к субъекту - возможные операции чтения, от субъекта к объекту - записи. Достижимость (возможность пройти по диаграмме, следуя стрелкам) одной вершины-субъекта с другой вершины-субъекта означает потенциальную возможность передачи данных от одного к другому, хотя бы и не напрямую. Достижимость вершины-объекта с другой вершины-объекта как раз и отражает информационный поток от одного объекта (класса объектов) к другому (см. [рис. 9.3](http://www.excode.ru/art6280p1.html#image.9.3)).

**Рис. 9.3.** Горизонтальные информационные потоки: диаграмма достижимости



На приведенном примере **прямая** передача информации из объекта "о1" в объект "о3" невозможна, однако существует информационный поток "о1--с1--о2--с3--о3", организуемый субъектами "с1" и "с3". Невозможна и прямая передача данных от субъекта "с3" субъекту "с1", однако через "третьи руки" это можно сделать: "с3--о3--с2--о1--с1". Алгоритмы выявления достижимости достаточно просты, однако они не дают рекомендации по тому, как **изменить** существующую модель, чтобы избежать нежелательных информационных потоков.

Механизм предоставления доступа может быть **статическим**, при котором решение о предоставлении доступа система принимает заранее, на основании свойств субъекта и объекта, или **динамическим**, когда на результат влияет текущее состояние субъекта, объекта, а также состояние всей системы. В частности, могут использоваться данные подсистемы учета ресурсов или состояния некоторых иных объектов и субъектов системы. Полная диаграмма достижимости при динамическом доступе может показывать **ложные** информационные потоки, опирающиеся на невозможные переходы в состояниях системы, но может и выявлять **скрытые** каналы переброски данных. Таким скрытым каналом может быть, например, **отсроченная** передача данных, когда один субъект записывает информацию, а другой, при определенном стечении обстоятельств, впоследствии получает доступ к ней.

Динамическим можно назвать правило отказывать в доступе, если в системе активен некоторый привилегированный субъект, или механизм "жетонов" для одноразового доступа к объекту. Когда используется статический механизм, говорят о правах доступа, когда динамический или смешанный - о сеансах доступа. Правила организации сеансов доступа могут быть весьма запутанными, потому что решение, которое примет система, зависит от состояния любой ее части. Такие правила привязаны к внутреннему устройству системы и реализации ее объектов. Пример динамических механизмов предоставления доступа см. в [[20]](http://www.excode.ru/popup.lit.html#20) и [[12]](http://www.excode.ru/popup.lit.html#12).

Организация прав доступа может быть целиком отдана на откуп самой системе, точнее, доверенному субъекту, на действия которого распространяются **не все** правила, определяющие доступ к объектам. В этом случае доверенный субъект (как правило, администратор системы) составляет некий свод правил, с которыми система и будет сообразовываться в дальнейшем. Назовем это централизованной ответственностью за потоки информации.

Другая возможность - так называемая доменная ответственность. Вводится понятие хозяина объекта. Хозяин объекта - это субъект, определяющий права доступа других субъектов к данному объекту. Таким образом, любой объект системы попадает в зону ответственности (домен) некоторого субъекта. Другой субъект, имеющий право читать данные из этого объекта, возможно, поместит их в другой объект, уже в своем домене. Тогда к этим данным, возможно, получит доступ третий субъект; так информация перетекает из домена в домен, направляемая хозяевами объектов.

#### Субъект-субъектная модель

Как система определяет хозяина объекта? Самый естественный способ - снабдить объект ярлыком, в котором будет написано имя хозяина. Субъект, чьим ярлыком подписан объект, устанавливает права доступа к нему всех нехозяев. При этом соблюдается принцип собственности: есть моя вещь, что хочу с ней, то и делаю; а есть чужая вещь, с ней можно делать только то, что позволит хозяин. Назовем такую стратегию субъект-субъектной моделью прав доступа. В самом деле, объект в этой модели не имеет самостоятельного места, все вопросы о доступе решаются на уровне двух субъектов: хозяина и пользователя.

Простая субъект-субъектная модель вполне работоспособна, особенно если считать, что у объекта может быть несколько хозяев - действительных субъектов, объединенных одним ярлыком. С точки зрения подсистемы, разделяющей доступ, им должен соответствовать один номинальный субъект - **группа**. Если же действительный субъект системы может принадлежать **более чем одной** группе, говорят о модели со множественным субъектом. Это более гибкий способ организовать права доступа на субъект-субъектном уровне. Невыполнимая в случае простой модели задача "запретить доступ конкретного субъекта к конкретному объекту" с помощью групп решается в три приема. Надо завести новую группу и добавить в нее всех субъектов-хозяев из старой группы, которой принадлежит объект. Всех, кроме одного: он-то и будет ущемлен в правах, когда мы пометим объект как принадлежащий новой группе.

С одной стороны, субъект-субъектная модель со множественным субъектом замечательно выражает общепринятое представление о правах собственности. С другой стороны, некоторые более или менее понятные изменения прав доступа к отдельному объекту выполняются непросто и, строго говоря, необратимо: изменив описанным в примере способом права доступа пользователя к объекту, мы теряем исходный ярлык этого объекта (теперь он принадлежит вновь созданной группе). Если запрет на доступ - временный, то когда-нибудь придется "все вернуть на место", а сделать это простым удалением новой группы и возвратом старого ярлыка нельзя еще и потому, что за время действия запрета объект мог еще раз поменять ярлык (например, еще кому-то запретили им пользоваться). Поэтому для **обратного** действия все равно придется **создавать** новую группу и включать в нее всех хозяев объекта плюс восстановленного в правах пользователя.

#### Субъект-объектная модель

Такая ситуация (как и любая, когда речь идет о доступе к объекту независимо от того, кому он принадлежит) воспринимается как исключение из правил - не такое уж редкое, впрочем. Здесь требуется иной подход к организации прав доступа, субъект-объектная модель, которая описывает взаимоотношения между субъектом и объектом.

Простая субъект-объектная модель подразумевает нечто вроде таблицы, в которой по горизонтали перечислены субъекты системы, а по вертикали - объекты; на пересечении горизонтали и вертикали вписаны права доступа соответствующего субъекта к соответствующему объекту. Такая таблица хороша, только если количество субъектов, объектов и способов доступа к ним в системе невелико и редко меняется (хотя сами права доступа могут изменяться часто). В самом деле, количество записей в таблице равно **произведению** этих трех системных величин, при этом, чтобы добавить туда новый объект, нужно определить права доступа к нему **каждого** субъекта. Это, во-первых, может потребовать немало системных ресурсов, а во-вторых, требует от того, кто добавляет, знания всего списка пользователей и их полномочий.

Введение прав доступа к объекту **по умолчанию** может спасти от второй напасти, но упомянутая таблица рассредоточивается, а зачастую становится лишь воображаемой. В таком варианте каждый объект носит за собой неопределенной длины "хвост", в котором заданы права доступа некоторых субъектов и отдельные - "для остальных". Такой "хвост" носит название таблицы управления доступом (Access Control List, ACL). Однако от первой напасти - несоразмерности служебных и полезных данных - механизм ACL спасает лишь отчасти: дублированной информации становится меньше, но в любом случае **максимальный** объем служебных данных при каждом объекте пропорционален количеству субъектов. Значит, ограничивать **максимальный** объем служебных данных в системе вообще нельзя, ведь при добавлении каждого нового субъекта этот объем будет - теоретически - возрастать на несколько процентов.

Не помогает и введение в модель множественного субъекта, хотя опять-таки сокращается объем дублированной информации. Введение в модель множественных **объектов**, групповых операций по изменению ACL или механизма наследования, когда дочерний объект подчиняется ACL родительского (например, файл наследует ACL каталога, в котором находится), только запутывает дело. Во-первых, как в случае субъект-субъектной модели, начинают возникать ситуации, когда разрешить (или запретить) доступ невозможно без пересмотра всей структуры прав доступа. Во-вторых, процедура определения фактических прав доступа (можно-таки пользователю читать файл или нет?) начинает зависеть от структуры хранилища объектов (файловой системы) и становится донельзя сложной и непонятной.

Между тем основные потребности в ограничении доступа покрываются именно субъект-субъектной моделью. В ней объем служебной информации об объектах **не зависит** от количества субъектов (к каждому объекту добавляется ярлык и фиксированный список прав доступа). Поэтому довольно удачной представляется субъект-субъектная модель прав доступа, дополненная ACL в качестве механизма обработки исключений. Скорее всего, случаи, для которых необходим ACL, будут редкими и особенного разрастания системной области не произойдет. Если, паче чаяния, объем ACL при разнообразных объектах системы станет слишком велик, это будет сигналом того, что само разбиение на группы доступа сделано администратором нерационально.