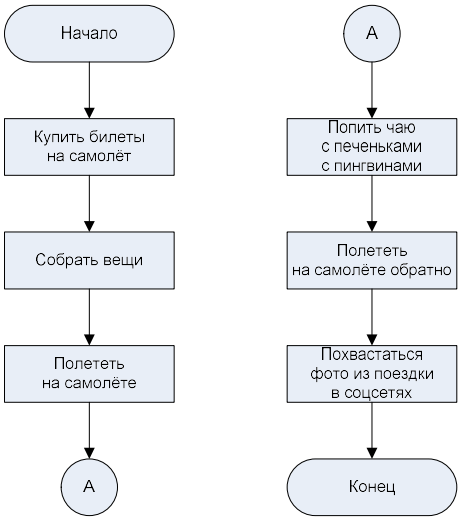
ТУБ №10:
Нисходящее и восходящее проектирование

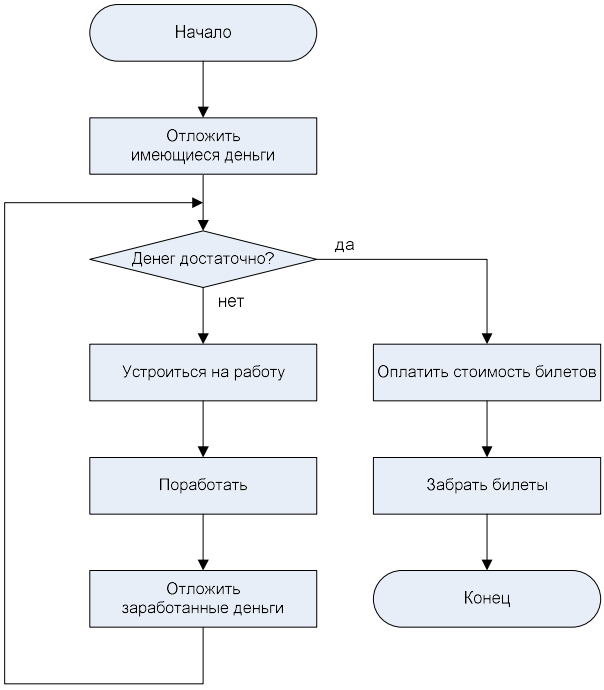
Линейные, разветвляющиеся и циклические вычислительные процессы — это основа любой современной программы. Но между небольшим циклом с параметром и программой, которая обеспечивает работу автопилота в условном Boeing 737, очевидно что-то пропущено. Что-то очень важное.

Редкий алгоритм, решающий целиком какую-либо задачу, будет в чистом виде линейным, разветвляющимся или циклическим — как правило, алгоритмы имеют смешанный характер и состоят из других, более простых алгоритмов.

Возьмём, например, задачу планирования поездки в Антарктиду. В первом приближении алгоритм линейный:



Но что там у нас насчёт покупки билетов?



Хьюстон, у нас проблема! Кажется, поездка откладывается… Тут целый цикл нарисовался. А если мы сейчас копнём поглубже, то окажется, что «Устроиться на работу» — это тоже цикл, в котором тоже некоторые этапы окажутся сложнее, чем могло бы показаться. А потом ещё при оплате билетов сколько нюансов будет, и это мы ещё даже не вылетели!

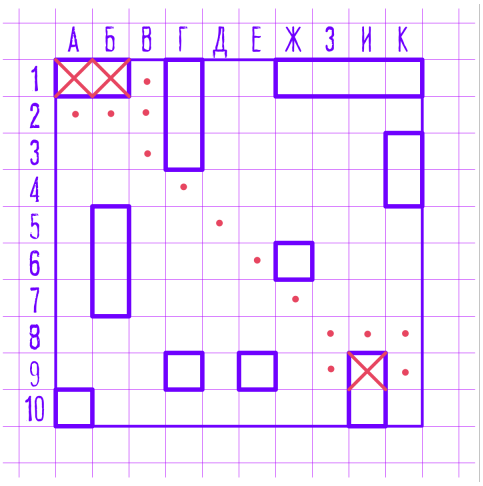
Критически настроенный читатель скажет, что в Антарктиду можно и на корабле добраться, но морская вода воздуха не слаще: всё равно решение задачи будет очень непростым. Тем не менее, если довести дело до конца, разобравшись с каждый шагом нашего первоначального алгоритма до уровня простых действий, то получится очень подробная пошаговая инструкция, который действительно позволит побывать в Антарктиде каждому, кто не боится длинных инструкций.

В программировании часто применяется аналогичный подход: сложная задача разбивается не более мелкие подзадачи, те в свою очередь на ещё более мелкие — и так до тех пор, пока не удастся дойти до уровня элементарных действий, которые уже могут быть записаны на каком-либо языке программирования. У этого подхода даже есть своё название — нисходящее проектирование.

Но ведь если есть нисходящее, значит, должно быть и восходящее? Именно так!

Допустим, нам нужно разработать игру. Скажем, «Морской бой». Пожалуй, здесь сложно сразу сказать, из каких крупных блоков будет состоять в итоге программа, поэтому начнём с самых простых вещей. Во-первых, очевидно, что нужно будет уметь рисовать на экране квадраты: из них состоят изображения игровых полей, ими же можно будет изображать и корабли. Во-вторых, для обозначения результатов выполненных игроками ходов понадобятся также алгоритмы рисования крестика и точки. В-третьих, надо будет подписать строки и столбцы.

В какой-то момент, разрабатывая эти относительно простые алгоритмы, мы заметим, что теперь можем решить более сложную задачу: нарисовать игровое поле целиком:



Ещё нам понадобятся разные алгоритмы, позволяющие воспроизводить звуки, реагировать на действия пользователей и т.д. Их мы тоже можем построить, начиная с самых базовых вещей и постепенно наращивая сложность. Рано или поздно мы придём к тому, что у нас будут в наличии все алгоритмы, необходимые для построения игры целиком. Этот подход как раз и называется восходящим проектированием.

Но какой из этих подходов лучше? На этот вопрос однозначного ответа нет.

С одной стороны может показаться, что лучше всё-таки восходящее проектирование: пусть у нас нет сразу готовой программы, но мы разрабатываем несложные алгоритмы и имеем возможность сразу их протестировать. Получается, на каждом этапе у нас используются составные части, в которых мы уверены и которые можем увидеть в действии.

В то же время в попытке угадать, что нам понадобится для разработки итоговой программы, можно разработать очень много алгоритмов, которые в итоге окажутся ненужными. Просто потому что нам казалось, что они будут нужны, но при дальнейшей разработке удалось обойтись без них. А это значит, что было потрачено лишнее время. Конечно, можно потом использовать эти лишние алгоритмы в других программах, но какие-то могут так и не пригодиться, да и на разработку этой конкретной игры времени уже потрачено больше.

Может показаться, что нисходящее проектирование лишено этих недостатков. Действительно, ведь на каждом шаге мы разбиваем какую-либо задачу на более мелкие подзадачи и для решения всей задачи каждая из этих подзадач должна быть решена. Но вот проблема: не всегда удаётся выполнить это разбиение (это ещё называют словом «декомпозиция») оптимальным способом. И если мы не угадали, то вместо одного относительно простого алгоритма придётся разрабатывать два более сложных, тесно связанных между собой.

К тому же, всегда есть риск разделить задачу на две части — простую и невозможную. Например, мы хотим превратить корову в страуса. Казалось бы, нет ничего проще: разделяем корову на молекулы, собираем из них страуса, оживляем то, что получилось, сэкономленные молекулы пускаем на благотворительность — готово! Но житейский опыт подсказывает, что фарш невозможно провернуть назад да и с молекулами это тоже не получится так уж просто, уж больно они непослушны. Что же до оживления — так это вообще мало кому удавалось.

В общем, серебряной пули нет, поэтому программисты при разработке сложных программ обычно комбинируют оба подхода: какие-то части разрабатывают восходящим способом, какие-то — нисходящим. Разумеется, с учётом их сильных и слабых сторон, которые мы только что обсудили.