TACHBIT · КУРС НА 30 ДНЕЙ · ЧАСТЬ 1

Как думает программа

Часть 1. Дни 1–14: основа логики и как устроены приложения. 14 паттернов, по одному в день, с проверенным кодом на Tachbit.

Приложение
ждёт события

Игра
живёт 60 кадров в секунду

ОС
служит всем остальным

Версия от 25 ноября 2026 · Код проверен компилятором Tachbit 1.3

Честно: готов ли Tachbit

Состояние на 25 ноября 2026 года (Tachbit 1.3).

> **Коротко:** учиться и делать свои приложения, 2D-игры и простые 3D-игры уже можно. Свою ОС можно собирать и запускать в эмуляторе. 3D-игры со светом, тенями и готовыми моделями тоже можно. Ставить Omnium на ноутбук пока рано: ОС не проверена на живом железе.

Что	Статус	Что уже есть	Чего пока нет
Приложения с окнами	Готово	Окна (и несколько окон сразу), кнопки, поля ввода, списки, меню, вкладки, перетаскивание (`lib/ui.tb`), файлы, JSON, HTTP, дата, звук, буфер обмена, чёткий шрифт на Retina. Пример — файловый менеджер `examples/apps/files` .	Готовых библиотек пока мало.
2D-игры	Готово	Игровой цикл с `frame_ms()` , клавиатура, мышь и геймпад, рисование, звук и свой микшер, плавность (`lib/anim.tb`), физика (`lib/physics.tb`).	Музыка в OGG пока не читается, только WAV.
3D-игры	Средние	Сетки, камера, поворот во все стороны, солнце, лампы, туман, тени, стекло, кости, готовые модели OBJ/glTF (`lib/model.tb`), свои шейдеры на Tachbit (`gpu func`), большие миры (LOD). Всё считает видеокарта.	Редактора сцен нет: мир собирается кодом. Часть шейдеров на Mac ещё не проверена.
Своя ОС (Omnium)	Работает в эмуляторе	64-битное ядро, несколько ядер процессора, клетки и каналы, память с квотами, диски SATA и NVMe, файловые системы OFS3 и exFAT, экран 1920×1080, мышь, оболочка «Капля», картинки JPEG/GIF/BMP.	Проверено только в QEMU, на настоящем компьютере не запускалось. Сети пока нет.
Системные программы на Linux/Mac	Готово	Ручная память (`danger`), раскладка байтов (`shape`), C-библиотеки (`use c`), системные вызовы, атомарные счётчики, сборка своей библиотеки для C.	Нужна аккуратность: в `danger` компилятор помогает меньше.

Почему курс про логику, а не про синтаксис

Синтаксис Tachbit можно выучить за пару недель по `docs/guide.md` . Труднее понять, *как думает* программа: где она хранит данные, кто их меняет, в каком порядке всё происходит. Эти приёмы называют **паттернами**. Они одинаковы в любом языке, поэтому всё, что вы здесь выучите, пригодится и в других языках.

Каждый день разбирает один паттерн и показывает, где он живёт в приложении, в игре и в операционной системе. Код в курсе проверен компилятором Tachbit, а вывод под кодом получен настоящим запуском.

Большая картина

Приложение, игра и ОС выглядят совсем разными, но внутри у них один скелет. Есть **состояние** (что программа помнит), **события** (что его меняет), **цикл** (кто ждёт события) и **слои** (кто за что отвечает). Разница только в том, как крутится цикл.

Приложение

Спит, пока ничего не происходит. Клик или сообщение будит его, оно меняет данные, перерисовывает окно и снова засыпает.

Игра

Не спит никогда. 60 раз в секунду: сколько прошло времени, что нажато, сдвинуть мир, нарисовать кадр заново.

Операционная система

Служит другим программам: делит между ними процессор, память, диск и экран, следит, чтобы никто не мешал другому.

Цикл приложения

ждать событие → разобрать → изменить состояние → перерисовать ↺

Цикл игры

dt → ввод → обновить мир → нарисовать кадр ↺ 60 раз в секунду

Цикл ядра ОС

прерывание → кто это: таймер, клавиша, диск → разбудить нужную службу → выбрать следующую задачу ↺

План на 30 дней

Дни 1–7	**Основа логики** Состояние, решения, циклы, модели, функции, ошибки. Без этого не работает ни одна программа.
Дни 8–14	**Как устроены приложения** Справочники, сохранение, слои, цикл событий, модель и вид, автоматы, отмена.
Дни 15–22	**Как устроены игры** Игровой цикл, физика, столкновения, сущности, экраны, плавность, события, ИИ.
Дни 23–30	**Как устроена система** Службы, память, параллельность, очереди, буферы, планировщик, права, общий скелет.

Как заниматься

Около часа в день. Прочитайте объяснение и наберите код руками: не копируйте его, так логика запоминается лучше. Запустите `tb run dayNN.tb` и сравните вывод с тем, что в книге. Потом сделайте задание. Если задание не получилось, это нормально: вернитесь к нему через день.

Готовые файлы примеров лежат рядом с этой книгой в архиве `tachbit-kurs-kod.zip`. Курс разделён на две книги: в первой части дни 1–14 (логика и приложения), во второй дни 15–30 (игры и система).

1 Состояние: что программа помнит

Неделя 1

Цель: Понять, что любая программа — это данные, которые меняются по шагам.

Любая программа что-то помнит: сколько денег на счёте, где стоит герой, какие окна открыты. Всё, что программа помнит прямо сейчас, называют **состоянием**. Каждая строка кода либо читает состояние, либо меняет его.

Если вы в любой момент можете сказать, какое у программы состояние, вы её понимаете. Большинство ошибок — это когда состояние стало не тем, что вы думали. Поэтому первый навык: держать в голове список того, что программа помнит.

ПРИЛОЖЕНИЕ
Корзина магазина: список товаров и сумма — это состояние. Кнопка «добавить» меняет его.

ИГРА
Монеты, жизни, позиция героя, номер уровня. Каждый кадр немного меняет эти числа.

ОС
Список запущенных программ, сколько памяти у каждой, какой файл открыт кем.

Паттерн «Состояние». Сначала выпишите, что программа помнит. Код — это правила, как это меняется.

КОД: DAY01.TB

```
// день 1: программа = данные + шаги
func main()
    coins = 0            // состояние: что программа помнит
    lives = 3
    coins += 10          // шаг меняет состояние
    lives -= 1
    print("монеты:", coins, " жизни:", lives)
```

ВЫВОД

```
монеты: 10  жизни: 2
```

Задание. Добавьте переменную `level`. Когда монет стало 10 или больше, увеличьте уровень на 1 и выведите его.

2 Решения и «охранник»

Цель: Научиться отсекать плохие случаи в начале функции.

Программа постоянно решает: можно или нельзя, туда или сюда. Новички пишут вложенные `if` в пять этажей, и логика теряется. Профессионалы делают наоборот: сначала проверяют всё, что может пойти не так, и сразу выходят. Это и есть **охранник** (guard).

После охранников остаётся «счастливый путь»: код, который работает, когда всё в порядке. Его легко читать, потому что он не спрятан внутри условий.

ПРИЛОЖЕНИЕ
Вход на сайт: нет логина — выход; неверный пароль — выход; заблокирован — выход; дальше вход.

ИГРА
Прыжок: герой в воздухе — нельзя; оглушён — нельзя; иначе прыгает.

ОС
Открыть файл: нет такого — ошибка; нет прав — ошибка; занят — ошибка; иначе открыть.

Паттерн «Охранник». Плохие случаи проверяй первыми и выходи сразу. Хороший путь — последним, без вложенности.

КОД: DAY02.TB

```
// день 2: решения и «охранник»
func can_buy(money: int, price: int, age: int) gives ok: bool
    ok = false
    if age < 18
        return           // охранник: плохой случай уходит сразу
    if money < price
        return
    ok = true

func main()
    print(can_buy(500, 120, 20))
    print(can_buy(100, 120, 20))
    print(can_buy(500, 120, 15))
```

ВЫВОД

```
true
false
false
```

Задание. Добавьте четвёртый параметр `banned: bool` и охранник для него. Проверьте три вызова.

3 Повторение и «накопитель»

Неделя 1

Цель: Пройти по списку и собрать из него итог: сумму, лучший, количество.

Почти любой отчёт, счёт или статистика устроены одинаково: завели пустой итог, прошли по всем элементам, каждый добавили в итог. Это **накопитель**.

В одном проходе можно копить сразу несколько итогов. Главное — правильное начальное значение: для суммы 0, для счётчика 0, для минимума — что-то очень большое.

ПРИЛОЖЕНИЕ
Отчёт за месяц: сумма продаж, лучший менеджер, сколько сделок больше 1000.

ИГРА
Итоги уровня: сколько врагов побеждено, лучший комбо-удар, общее время.

ОС
Сколько памяти занято всеми программами, какая программа ест больше всех.

Паттерн «Накопитель». Итог = начальное значение, затем один проход, каждый элемент меняет итог.

КОД: DAY03.TB

```
// день 3: повторение и «накопитель»
func main()
    scores = [12, 40, 7, 33, 25]
    total = 0
    best = 0
    many = 0
    each s in scores
        total += s               // сумма
        best = max(best, s)      // лучший
        if s ⩾ 25
            many += 1            // счётчик по условию
    print("сумма", total, " лучший", best, " от 25 и выше:", many)
```

ВЫВОД

```
сумма 117  лучший 40  от 25 и выше: 3
```

Задание. Добавьте `worst` — самый маленький результат. Подумайте, с какого числа его начать.

4 Фильтр, преобразование, свёртка

Цель: Разложить обработку списка на три понятных шага.

Работу со списками почти всегда можно разбить на три кирпичика. **Фильтр** оставляет только нужные элементы. **Преобразование** (map) меняет каждый элемент по правилу. **Свёртка** (fold) превращает много элементов в одно значение.

Когда задача кажется сложной, спросите себя: что я отбираю, что я меняю, во что я сворачиваю? Ответы и есть три цикла. Их потом легко проверять по отдельности.

ПРИЛОЖЕНИЕ
Счёт клиенту: отобрать неоплаченные услуги, к каждой добавить налог, сложить.

ИГРА
Взрыв: отобрать врагов в радиусе, каждому посчитать урон, сложить очки.

ОС
Отобрать процессы одного пользователя, взять у каждого память, сложить.

Паттерн «Фильтр → преобразование → свёртка». Отбери, измени каждый, сверни в одно. Каждый шаг — отдельный цикл.

КОД: DAY04.TB

```
// день 4: фильтр, преобразование, свёртка
func main()
    prices = [120, 900, 45, 300, 1500]
    cheap = list int
    each p in prices
        if p < 500
            cheap.add(p)                  // фильтр: оставить нужные
    with_tax = list int
    each p in cheap
        with_tax.add(p + p * 12 / 100)    // преобразование: каждый по правилу
    sum = 0
    each p in with_tax
        sum += p                          // свёртка: много в одно
    print(cheap, with_tax, sum)
```

ВЫВОД

```
[120, 45, 300] [134, 50, 336] 520
```

Задание. Отберите цены дороже 200, сделайте скидку 10% каждой и найдите самую большую.

5 Модель: свой тип для вещи из жизни

Цель: Описать предмет из задачи как свой тип с полями.

Когда у вещи много свойств, держать их в отдельных переменных неудобно. Объявите тип: `obj Player` с полями. Теперь игрок — одна вещь, которую можно передать в функцию, положить в список, сохранить.

Хорошая модель — половина программы. Начинайте любой проект с вопроса: какие вещи есть в моём мире и что о каждой нужно помнить? Ответ превращается в `obj`.

ПРИЛОЖЕНИЕ
`Client` с именем, телефоном и списком заказов; `Order` с суммой и статусом.

ИГРА
`Player`, `Enemy`, `Bullet` — у каждого позиция, скорость, жизни.

ОС
`Task` с номером, состоянием и памятью; `File` с именем, размером и владельцем.

Паттерн «Модель». Каждая вещь из задачи — свой тип. Поля — только то, что действительно нужно помнить.

КОД: DAY05.TB

```
// день 5: модель — свой тип для вещи из жизни
obj Player
    name: text
    hp: int
    items: list text

func main()
    p = Player(name: "Нази", hp: 100)
    p.items.add("меч")
    p.items.add("зелье")
    p.hp -= 30
    print(p.name, "здоровье", p.hp, "вещи", p.items)
```

ВЫВОД

```
Нази здоровье 70 вещи [меч, зелье]
```

Задание. Опишите `obj Room` (название, список игроков, максимум мест) и добавьте в комнату двух игроков.

6 Чистые функции

Неделя 1

Цель: Писать функции, которые получают всё через параметры и только возвращают ответ.

Функция — это маленькая машина: на вход данные, на выход ответ. Если функция ничего не меняет вокруг и на одни и те же данные всегда даёт один ответ, её называют **чистой**. Её легко проверить: подал числа — сравнил ответ.

Делайте расчёты чистыми, а запись в файл, рисование и сеть держите в отдельных функциях. Тогда ошибку в расчёте можно найти без окна и без интернета.

ПРИЛОЖЕНИЕ
Расчёт налога, скидки, срока — чистые функции. Сохранение в базу — отдельно.

ИГРА
Урон по броне, расстояние между точками, следующая позиция — чистые функции.

ОС
Перевод адреса памяти в номер страницы, проверка прав — чистые функции.

Паттерн «Чистая функция». Всё нужное — в параметрах, результат — через return, ничего вокруг не меняется.

КОД: DAY06.TB

```
// день 6: функции — вход, выход, без сюрпризов
func area(w: int, h: int)
    return w * h

func price_of(w: int, h: int, per_meter: int)
    return area(w, h) * per_meter

func main()
    print("ковёр 3×4 стоит", price_of(3, 4, 250))
```

ВЫВОД

```
ковёр 3×4 стоит 3000
```

Задание. Напишите `delivery(km: int)`: до 5 км — 100, дальше +20 за каждый км. Проверьте на 3, 5, 10.

7 Ошибка — это тоже ответ

Неделя 1

Цель: Отличать «не получилось» от «получилось», не ломая программу.

В реальном мире всё ломается: файла нет, человек ввёл буквы вместо цифр, сеть пропала. В Tachbit функция сообщает о проблеме словом `fail`. Тот, кто вызвал, решает: передать дальше (`try`), подставить запасное значение (`or`) или обработать (`else err`).

Главное правило: ошибку обрабатывает тот, кто знает, что с ней делать. Функция чтения числа не знает, показать окно или записать в журнал, — она только честно сообщает «не вышло».

ПРИЛОЖЕНИЕ
Форма: неверная дата — подсветить поле красным, а не закрыть программу.

ИГРА
Нет файла сохранения — начать новую игру с запасными настройками.

ОС
Диск не ответил — повторить, потом сообщить программе, но не уронить всю систему.

Паттерн «Ошибка как значение». Кто не знает, что делать, — передаёт дальше. Кто знает — обрабатывает.

КОД: DAY07.TB

```
// день 7: ошибка — это тоже ответ
func parse_age(s: text)
    n = try to_int(s)
    if n < 0 or n > 150
        fail "возраст вне границ: " + n
    return n

func main()
    a = parse_age("25") or -1
    print("возраст", a)
    b = parse_age("300") else err
        print("не вышло:", err)
    c = parse_age("abc") or -1
    print("запасное значение:", c)
```

ВЫВОД

```
возраст 25
не вышло: возраст вне границ: 300
запасное значение: -1
```

Задание. Напишите `parse_price(s)`, которая не принимает отрицательные цены, и обработайте три случая.

8 Справочник: найти по ключу сразу

Неделя 2

Цель: Использовать словарь, когда нужно быстро найти запись по имени или номеру.

Искать в списке — значит перебирать всё подряд. Если записей тысячи, это медленно. **Словарь** (`dict`) находит запись по ключу сразу, как телефонная книга по имени.

Используйте словарь, когда вопрос звучит «дай мне X по такому-то имени/номеру». Перед чтением проверьте `.has`: ключа может не быть.

ПРИЛОЖЕНИЕ
Клиент по номеру телефона, товар по артикулу, настройки по названию.

ИГРА
Текстуры и звуки по имени, предметы инвентаря по коду, рекорды по игроку.

ОС
Процесс по номеру, файл по пути, драйвер по типу устройства.

Паттерн «Справочник». Если ищешь по ключу чаще одного раза — нужен словарь, а не перебор.

КОД: DAY08.TB

```
// день 8: справочник — найти по ключу сразу
func main()
    stock = dict text int
    stock["чай"] = 10
    stock["кофе"] = 4
    stock["чай"] -= 3
    if stock.has("сок")
        print("сок есть")
    else
        print("сока нет")
    each name in stock
        print(name, stock[name])
```

ВЫВОД

```
сока нет
чай 7
кофе 4
```

Задание. Сделайте словарь цен и функцию, которая считает стоимость заказа `["чай", "кофе", "чай"]`.

9 Загрузить, изменить, сохранить

Неделя 2

Цель: Понять, как программа переживает выключение.

Всё, что в переменных, исчезает при выходе. Чтобы помнить между запусками, программа читает данные из файла при старте и записывает при изменении. Это цикл **загрузить → изменить → сохранить**.

Два правила. Первое: при загрузке учитывайте, что файла может не быть (первый запуск). Второе: сохранение может сорваться — проверьте ошибку и скажите о ней.

ПРИЛОЖЕНИЕ
Заметки, настройки, черновик письма, база клиентов.

ИГРА
Сохранение прогресса, рекорды, настройки звука.

ОС
Файловая система делает то же для всех: журнал изменений, запись на диск, восстановление после сбоя.

Паттерн «Загрузить → изменить → сохранить». Нет файла — начни с пустого. Ошибку записи никогда не молчи.

КОД: DAY09.TB

```
// день 9: сохранение — загрузить, изменить, записать
func main()
    path = "notes.txt"
    old = read_file(path) or ""
    lines = list text
    if old ≠ ""
        lines = split(old, "\n")
    lines.add("заметка " + (len(lines) + 1))
    write_file(path, join(lines, "\n")) else err
        print("не сохранил:", err)
        return
    print("в файле заметок:", len(lines))
```

ВЫВОД

```
в файле заметок: 1
```

Задание. Храните в файле счётчик запусков программы и выводите его каждый раз.

10 Слои: данные, логика, вывод

Цель: Разделять программу на части, которые не мешают друг другу.

Когда расчёт, показ и хранение перемешаны в одной функции, любая правка ломает всё сразу. Разделите программу на **слои**: данные (что храним), логика (что считаем), вывод (как показываем). Каждый слой знает только о соседе снизу.

Проверка простая: можно ли заменить вывод в консоль на окно, не трогая логику? Если да — слои разделены. Большие части выносите в отдельные файлы и подключайте словом `use`.

ПРИЛОЖЕНИЕ
Таблица клиентов (данные), правила статусов (логика), страница сайта (вывод).

ИГРА
Мир и объекты (данные), физика и правила (логика), рисование и звук (вывод).

ОС
Драйвер диска (низ), файловая система (середина), окно «Файлы» (верх).

Паттерн «Слои». Нижний слой не знает о верхнем. Вывод только читает, логика только считает.

КОД: DAY10.TB

```
// день 10: слои — данные, логика, вывод отдельно
// файл shop_logic.tb рядом:  modpub func total(prices: list int) ...
obj Order
    items: list int

func total(o: Order)                    // логика: только считает
    sum = 0
    each p in o.items
        sum += p
    return sum

func print_order(o: Order)                        // вывод: только показывает
    print("товаров", len(o.items), "на", total(o), "сом")

func main()
    o = Order()                         // данные
    o.items.add(120)
    o.items.add(300)
    print_order(o)
```

ВЫВОД

```
товаров 2 на 420 сом
```

Задание. Добавьте `discount(o: Order)` в слой логики и покажите сумму со скидкой, не меняя `total`.

11 Цикл событий

Цель: Увидеть сердце любой интерактивной программы: ждать, разобрать, ответить.

Программа с окном или терминалом почти всё время ничего не делает — она **ждёт событие**: нажатие, строку, сообщение из сети. Пришло — разобрали, что это, ответили, снова ждём. Этот бесконечный круг называют **циклом событий**.

Внутри круга не должно быть долгих дел: пока вы считаете минуту, программа «зависла» для человека. Долгое отдают в фон (день 25).

ПРИЛОЖЕНИЕ
Каждое приложение с окном: клик → обработчик → перерисовать → ждать дальше.

ИГРА
Тот же круг, только он не ждёт, а крутится 60 раз в секунду (день 15).

ОС
Ядро ждёт прерывания: таймер, клавиатура, диск. Обработало — вернулось к ожиданию.

Паттерн «Цикл событий». Жди → разбери → ответь → снова жди. Внутри круга только быстрые дела.

КОД: DAY11.TB

```
// день 11: цикл событий — ждём, разбираем, отвечаем
func main()
    count_ok = 0
    repeat while true
        line = input() else err
            stop                          // ввод закончился
        cmd = trim(line)
        if cmd == "выход"
            stop
        else if cmd == "привет"
            print("и тебе привет")
            count_ok += 1
        else
            print("не знаю команду:", cmd)
    print("понял команд:", count_ok)
```

ВЫВОД (ВВЕЛИ: ПРИВЕТ / ЧТО / ПРИВЕТ / ВЫХОД)

```
и тебе привет
не знаю команду: что
и тебе привет
понял команд: 2
```

Задание. Добавьте команды `время` (вывести `date_now()`) и `счёт` (сколько всего команд было).

12 Модель и вид

Цель: Отделить то, что программа помнит, от того, как это нарисовано.

В программе с окном есть два мира. **Модель** — данные: сколько нажали, какие задачи. **Вид** — картинка на экране. Правило: события меняют модель, а рисование только читает модель и ничего не меняет.

Так проще всего избежать ситуации «на экране одно, а в данных другое». Окно каждый кадр рисуется заново по модели, поэтому оно всегда честное. Библиотека `lib/ui.tb` работает именно так.

ПРИЛОЖЕНИЕ
CRM: карточки в базе — модель, доска с колонками — вид. Перетащили карточку — поменяли статус в модели.

ИГРА
Мир игры — модель, кадр на экране — вид. Камеру можно двигать, мир от этого не меняется.

ОС
Оболочка рисует окна по списку окон в памяти; программа сама пикселей рабочего стола не трогает.

Паттерн «Модель / вид». События меняют модель. Рисование только читает модель.

КОД: DAY12.TB

```
// день 12: состояние отдельно, рисование отдельно
use ui

obj Model
    clicks: int

func main()
    try window_open(420, 240, "Счётчик")
    u = ui.start()
    m = Model()
    repeat while window_alive()
        ui.frame(u)
        // 1) событие меняет состояние
        if ui.button(u, 40, 120, 160, 44, "Нажми")
            m.clicks += 1
        if ui.button(u, 220, 120, 160, 44, "Сброс")
            m.clicks = 0
        // 2) рисование только читает состояние
        ui.big_text(u, 40, 40, "нажато: " + m.clicks, 3, rgb(240, 240, 240))
        ui.end_frame(u)
```

ЧТО БУДЕТ

```
Откроется окно. Запускайте на своём компьютере, где есть экран.
```

Задание. Добавьте кнопку «−1», которая не опускает счёт ниже нуля (вспомните охранника).

13 Конечный автомат

Неделя 2

Цель: Описывать поведение через режимы и разрешённые переходы.

Многие вещи живут в **режимах**: дверь закрыта, открыта или заперта. В каждом режиме допустимы свои действия. Запертую дверь нельзя открыть — сначала отпереть. Если описать это флажками `is_open`, `is_locked`, легко получить невозможное «открыта и заперта одновременно».

Конечный автомат решает это: один `enum` с режимами и одна функция «режим + действие → новый режим». Невозможные состояния просто нельзя записать.

ПРИЛОЖЕНИЕ
Заказ: новый → оплачен → отправлен → доставлен. Нельзя отправить неоплаченный.

ИГРА
Герой: стоит, бежит, прыгает, падает. Из прыжка нельзя снова прыгнуть.

ОС
Задача: готова, работает, ждёт, завершена. Планировщик двигает её только по разрешённым стрелкам.

Паттерн «Конечный автомат». Один enum на режим, одна функция переходов. Никаких пар флажков.

КОД: DAY13.TB

```
// день 13: конечный автомат — в каком режиме и куда можно перейти
enum Door
    closed
    open
    locked

func next(d: Door, action: text) gives n: Door
    n = d
    match d
        closed
            if action == "открыть"
                n = Door.open
            else if action == "запереть"
                n = Door.locked
        open
            if action == "закрыть"
                n = Door.closed
        locked
            if action == "отпереть"
                n = Door.closed

func main()
    d = Door.closed
    each a in ["открыть", "запереть", "закрыть", "запереть", "открыть", "отпереть"]
        d = next(d, a)
        print(a, "→", d)
```

ВЫВОД

```
открыть → open
запереть → open
закрыть → closed
запереть → locked
открыть → locked
отпереть → closed
```

Задание. Добавьте режим `broken`: из любого режима действие «выбить» ломает дверь навсегда.

14 Команды и отмена

Цель: Превратить действие в предмет, который можно сохранить и отменить.

Если каждое действие пользователя записать как объект «что сделали и с чем», получится **команда**. Команды можно сложить в список — это история. Взять последнюю и сделать наоборот — это **отмена** (Ctrl+Z).

Тот же приём даёт повтор действий, макросы и запись игры: достаточно сохранить список команд и проиграть его снова.

ПРИЛОЖЕНИЕ
Отмена в редакторе, журнал действий в CRM, «вернуть удалённое».

ИГРА
Повтор матча, отмена хода в шахматах, запись ввода для проверки.

ОС
Журнал файловой системы: сначала записать, что собираемся сделать, потом делать — после сбоя можно доделать или откатить.

Паттерн «Команда». Действие — это объект. История — список команд. Отмена — обратная команда.

КОД: DAY14.TB

```
// день 14: команды и отмена
obj Cmd
    kind: text      // "add" или "del"
    value: text

func do_cmd(doc: list text, c: Cmd)
    if c.kind == "add"
        doc.add(c.value)
    else
        last = doc.pop()

func undo(doc: list text, c: Cmd)
    if c.kind == "add"
        last = doc.pop()
    else
        doc.add(c.value)

func main()
    doc = list text
    done = list Cmd
    each w in ["раз", "два", "три"]
        c = Cmd(kind: "add", value: w)
        do_cmd(doc, c)
        done.add(c)
    print("было:", doc)
    undo(doc, done.pop())                // Ctrl+Z
    undo(doc, done.pop())
    print("после двух отмен:", doc)
```

ВЫВОД

```
было: [раз, два, три]
после двух отмен: [раз]
```

Задание. Сделайте «повтор» (Ctrl+Y): отменённые команды складывайте во второй список и умейте применить их снова.

Шпаргалка: паттерны 1–14

День	Паттерн	Правило
1	**Состояние**	Сначала выпишите, что программа помнит. Код — это правила, как это меняется.
2	**Охранник**	Плохие случаи проверяй первыми и выходи сразу. Хороший путь — последним, без вложенности.
3	**Накопитель**	Итог = начальное значение, затем один проход, каждый элемент меняет итог.
4	**Фильтр → преобразование → свёртка**	Отбери, измени каждый, сверни в одно. Каждый шаг — отдельный цикл.
5	**Модель**	Каждая вещь из задачи — свой тип. Поля — только то, что действительно нужно помнить.
6	**Чистая функция**	Всё нужное — в параметрах, результат — через return, ничего вокруг не меняется.
7	**Ошибка как значение**	Кто не знает, что делать, — передаёт дальше. Кто знает — обрабатывает.
8	**Справочник**	Если ищешь по ключу чаще одного раза — нужен словарь, а не перебор.
9	**Загрузить → изменить → сохранить**	Нет файла — начни с пустого. Ошибку записи никогда не молчи.
10	**Слои**	Нижний слой не знает о верхнем. Вывод только читает, логика только считает.
11	**Цикл событий**	Жди → разбери → ответь → снова жди. Внутри круга только быстрые дела.
12	**Модель / вид**	События меняют модель. Рисование только читает модель.
13	**Конечный автомат**	Один enum на режим, одна функция переходов. Никаких пар флажков.
14	**Команда**	Действие — это объект. История — список команд. Отмена — обратная команда.