Опыт создания визуальных редакторов бизнес-логики для непрограммистов на Lua и JavaScript

Александр Гладыш
LogicEditor.com, CTO
ag@logiceditor.com

Что такое «бизнес-логика»?

- Бизнес-логика — совокупность правил, принципов, зависимостей поведения объектов предметной области.
- Иначе можно сказать, что бизнес-логика — это реализация правил и ограничений автоматизируемых операций.

— *Wikipedia*

Что такое «бизнес-логика»?

http://www.flickr.com/photos/danisarda/2868413519/

Как пишется бизнес-логика?

Классика:

→ Придумали
→ Поставили в план программистам
→ Программисты добрались до тикета,
поругались, что-то написали
→ Потестировали, поняли, что всё не так
→ Поставили в план программистам доработки
→ ...и так ad nauseam

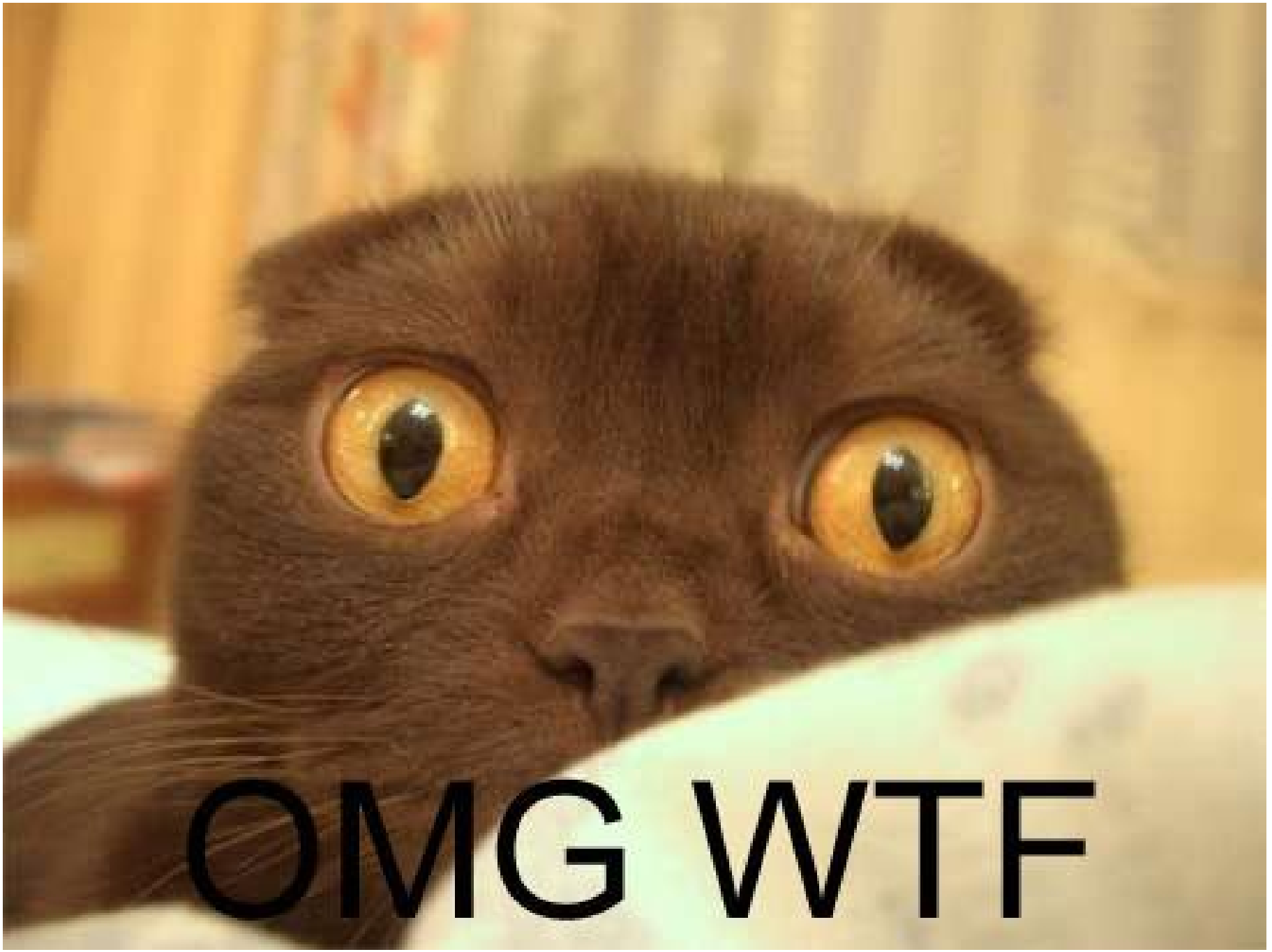
Как пишется бизнес-логика?

Выделенный программист:

→ Придумали
→ Объяснили задачу программисту
→ Программист «сразу» написал код
→ Потестировали, попросили поправить
→ ...и так ad nauseam

Как хочется писать бизнес-логику?

→ Сам придумал
→ Сам реализовал
→ Сразу протестировал
→ Поправил как надо
→ ...
→ PROFIT!!!



http://www.flickr.com/photos/buro9/2990026685

«Почему гейм-дизайнеры — не программисты?»

Профессиональные программисты, помимо навыков составления алгоритмов и их реализации ещё, **как минимум,** имеют:

- Технические знания об используемой системе
- Навыки отладки
- Навыки командной работы с кодом

Что делать?

- Требуется средство работы с бизнес-логикой, которое будет прощать ***технические*** ошибки, а, лучше, **не давать их совершать.**
- При этом это средство должно, по возможности, ***минимально*** ограничивать свободу творчества пользователя.

Типичные решения

- Пустить пользователя в базу
- Дать ему писать XML
- Сделать ему бэкофис — админку

Недостаточно гибко!

Рано или поздно любая попытка
абстрагирования логики в данные
растягивается слишком сильно
и лопается.

Нужен редактор бизнес-логики!

Редактируем всю «логику» (алгоритм),
а не только «данные» (параметры алгоритма)

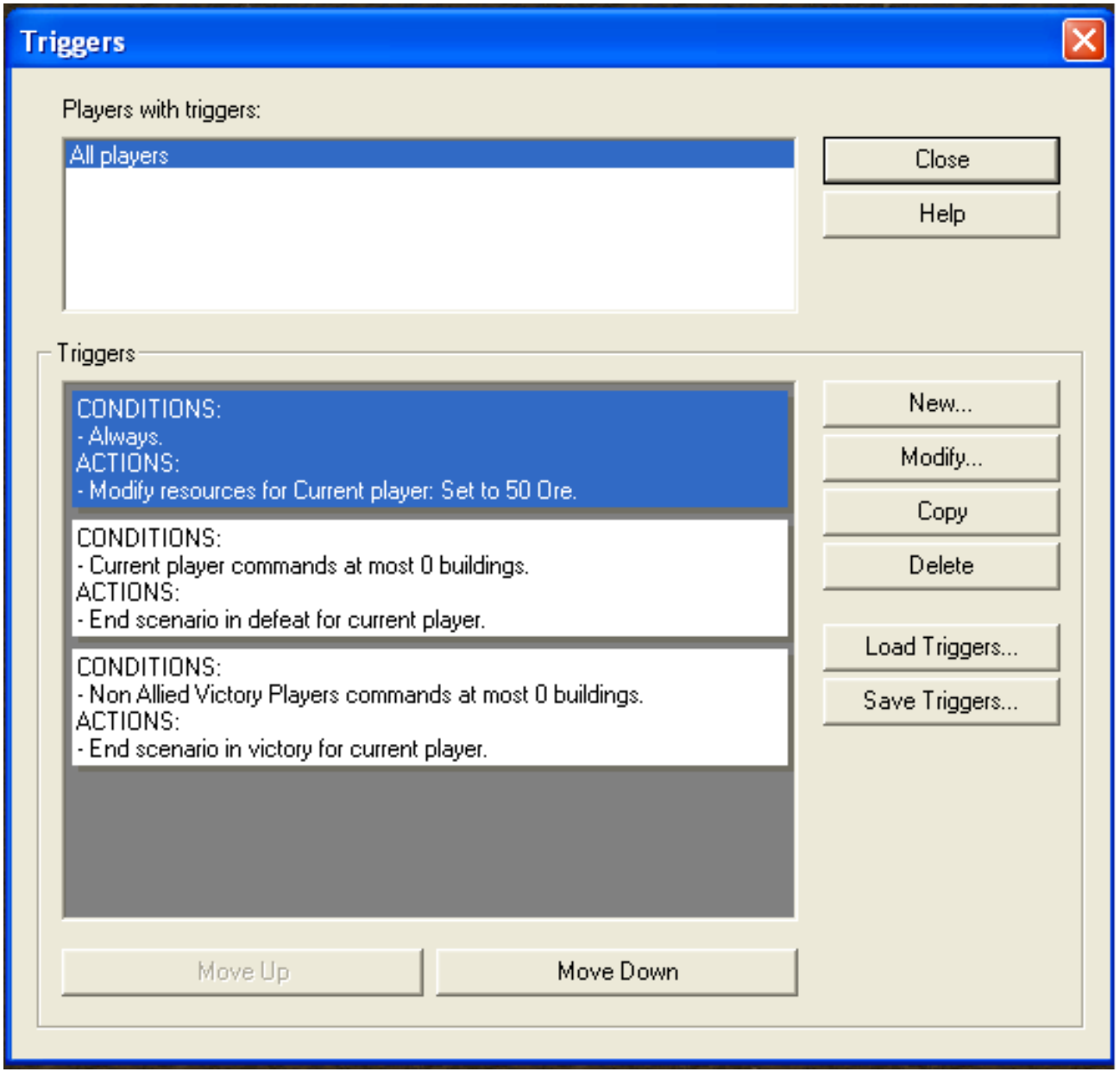
При этом нужно обеспечить:

- приемлемую кривую обучения;
- приемлемую сложность работы;
- доступную сложность разработки и поддержки, т. е. адекватную гибкость решения.

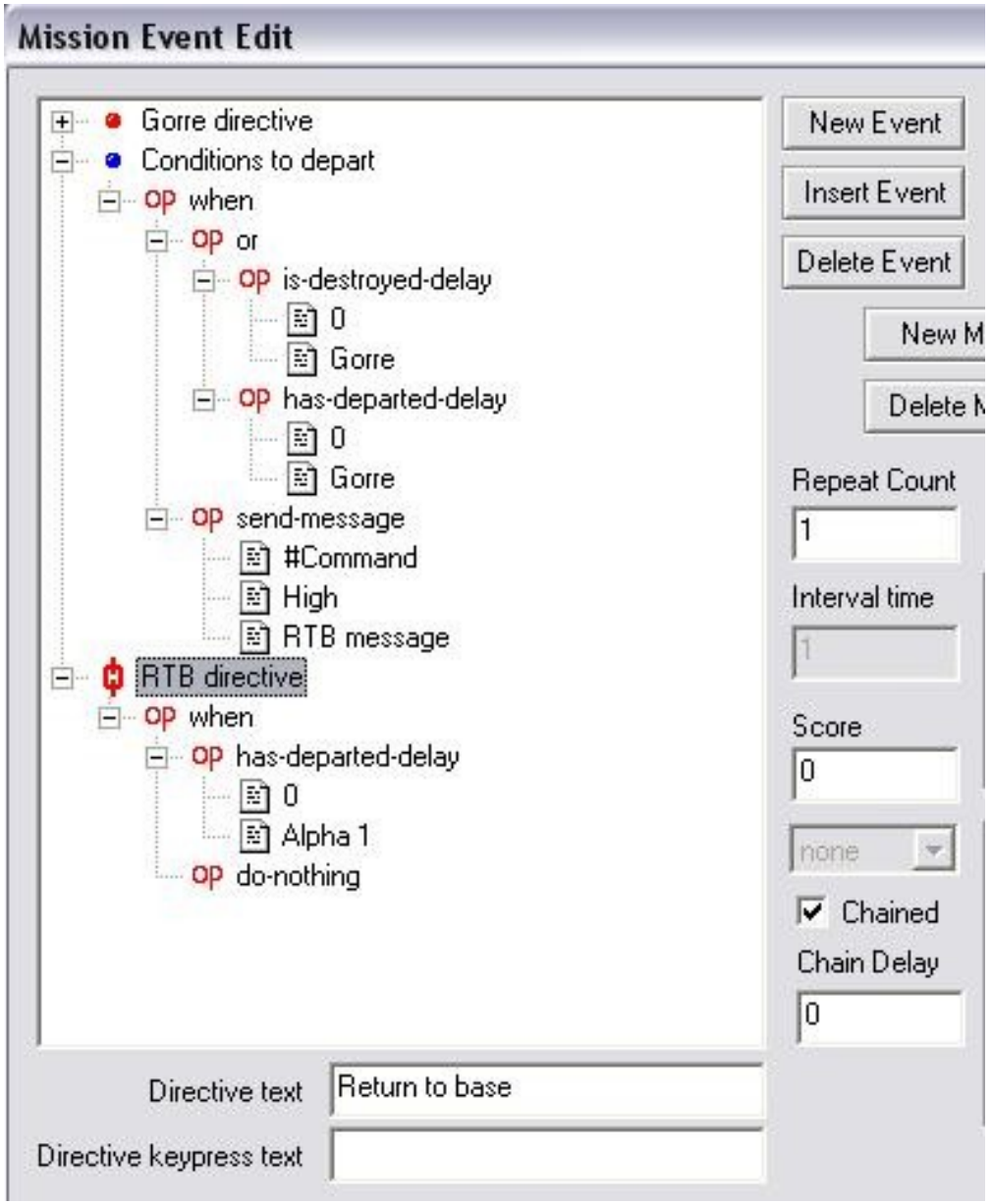
Как другие решают этот вопрос?

Несколько классических примеров

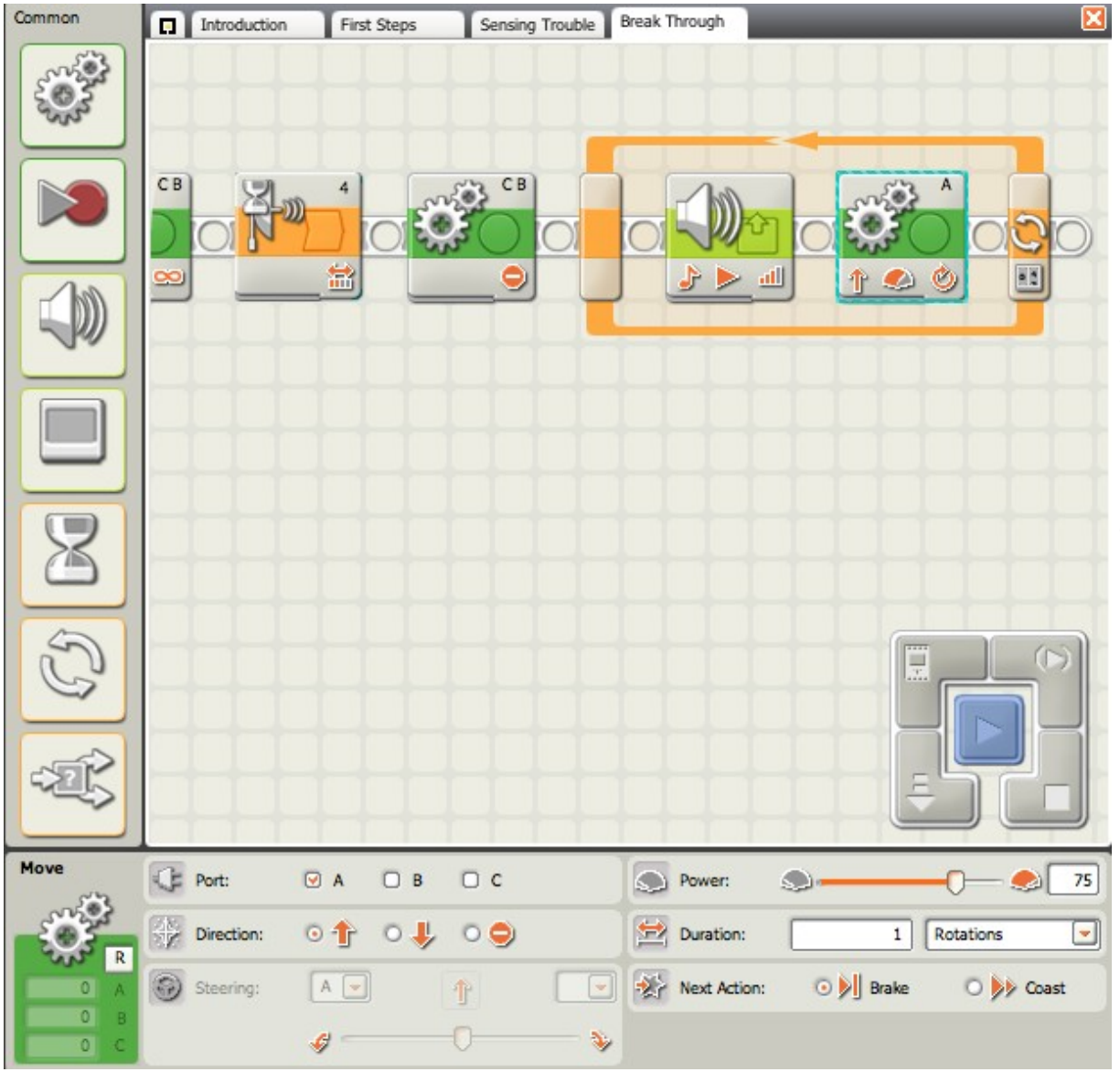
StarCraft Campaign Editor triggers



Descent Freespace Editor Events

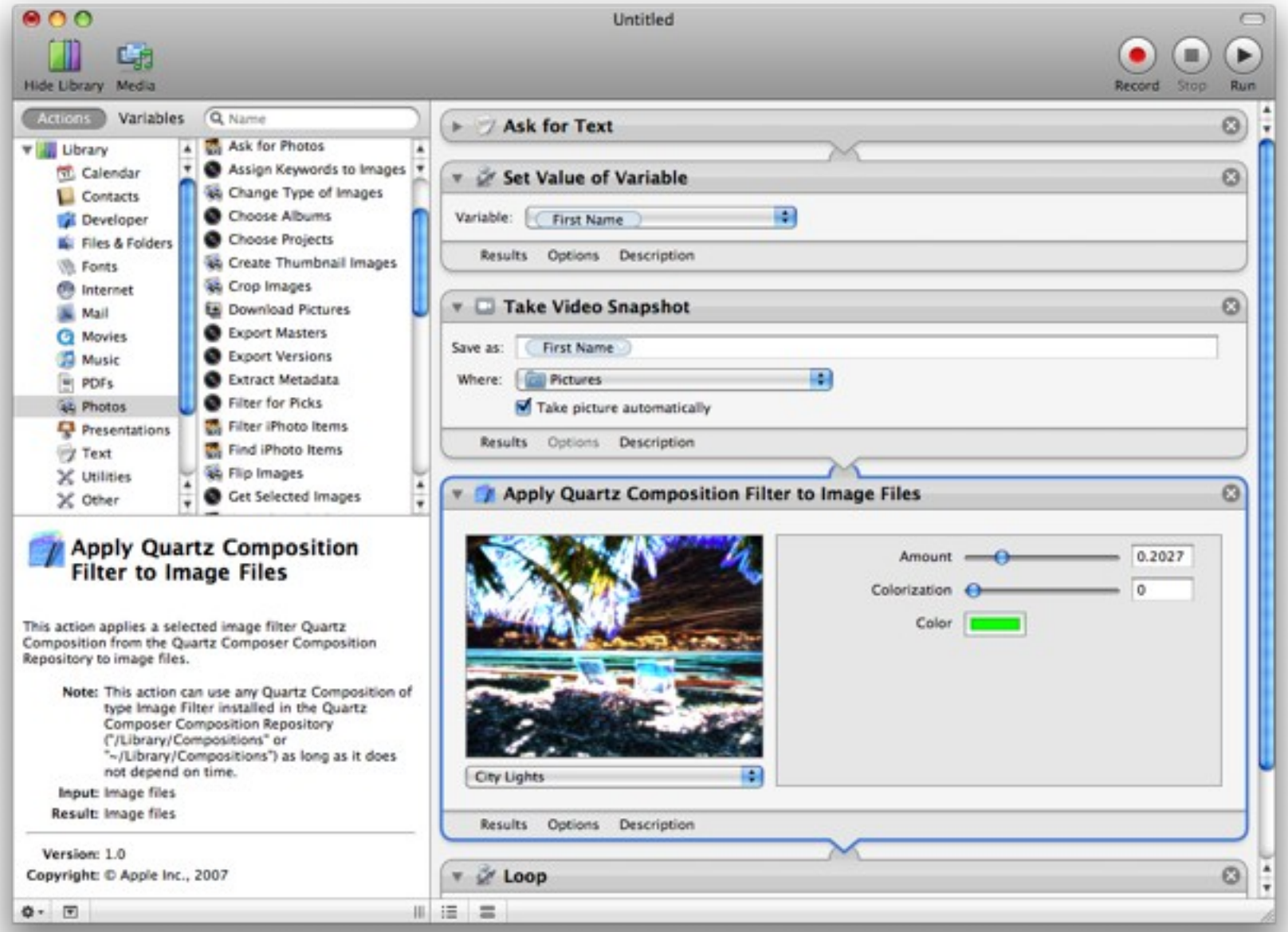


Lego NXT-G



http://mindstorms.lego.com/en-us/Software/

Apple Automator



http://www.macosxautomation.com/automator/index.html

Что общего?

Визуальный *DSL*,
описывающий
дерево *Control Flow*

Как это решали мы?

Краткая ретроспектива шести поколений идеи

В каждом случае реализация была сделана мной и / или моими коллегами полностью с нуля, включая графику, для разных заказчиков / работодателей. Повторно использовалась и развивалась только общая **идея** о том, как это должно работать.

Редактор видео-квестов

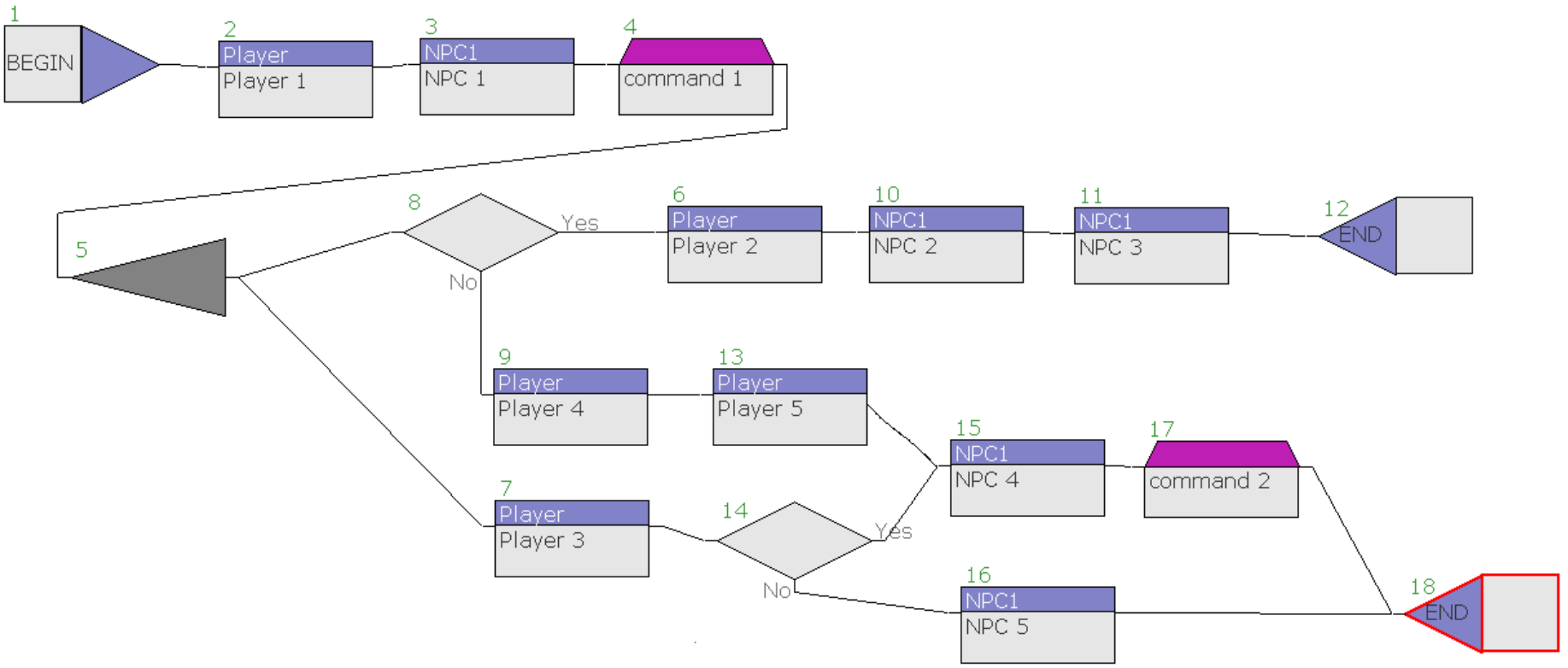
(Увы, скриншота не осталось)

Грубо:

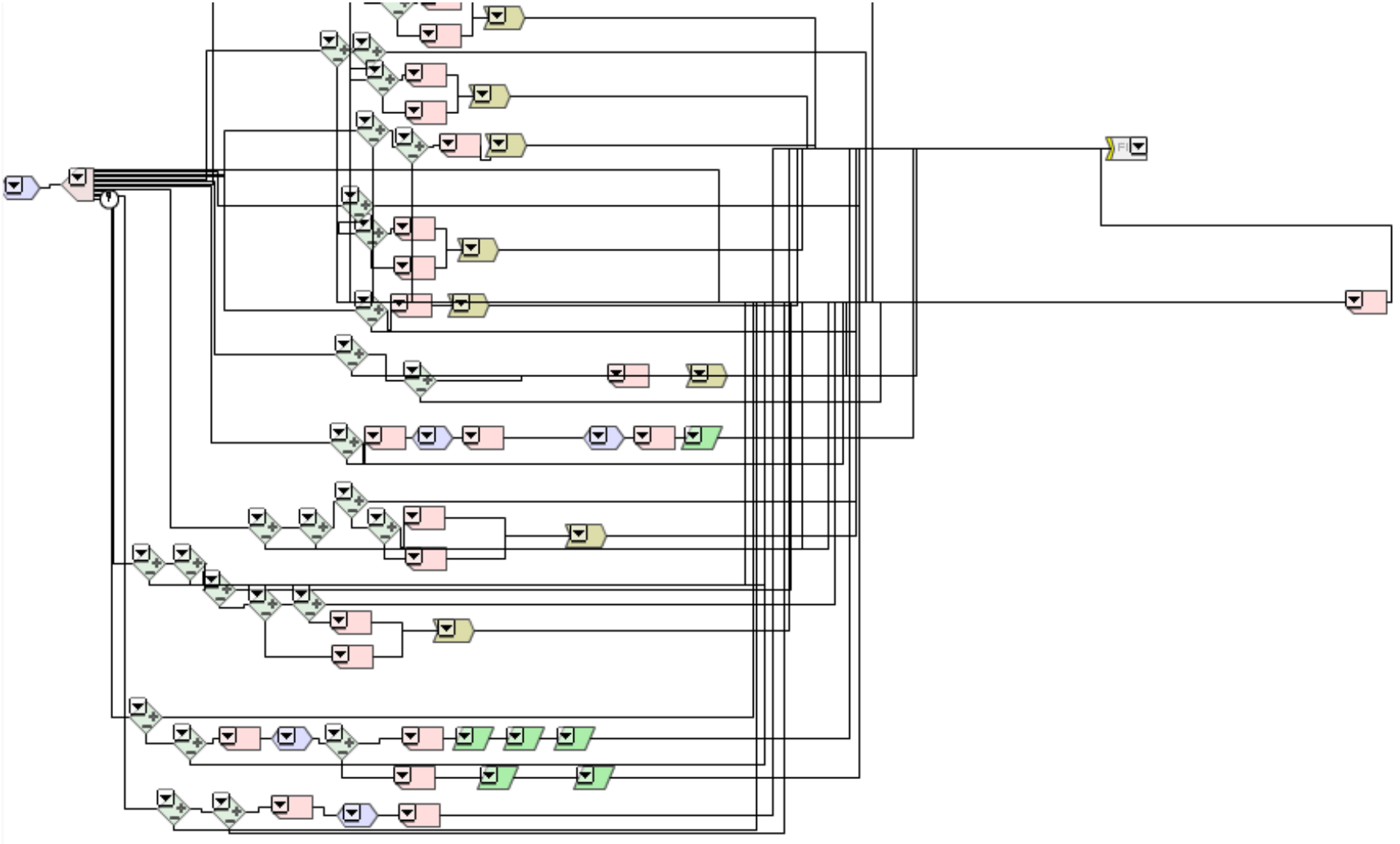
Граф из реплик диалогов с возможностью привязывать к ним видеоряд.

~2002–2004, достался в наследство

Редактор квестовых диалогов I, II



Редактор квестов I, II



Редактор магии

Цели

неинтерактивно[# x][+]

Мгновенные эффекты

Игнорировать активацию в статистике:ДА[#]

Действия:*Нет*[+]

Овертайм-эффекты

Цель: на себя[#]
Время жизни: 255[I] *(≥255 — бессрочно)*
Период: 0[I]
Изначальный кулдаун: 0[I]
Сброс в конце боя: НЕТ[#][B]
Остается при снятии всех эффектов вручную: ДА[#]
Максимальное число одновременно активных эффектов: 1[I] *(0 — не ограничено)*
Игровые режимы: дуэль[#]

При изменении набора характеристик

1. Если (изменения инициированы целью овертайм-эффекта[B ↓ x] **И** (жизнь[#] в наборе изменений противника[#][I] < уровней с 1[I] до 10[I] (учитывая уровень в счетчике: **ДА**[#][B])[I ↑ x][+])[I])[B ↑ x][+])[B], то
    1. Играть эффект абилки ID: 50402[I][A ↓ x]
    2. Активировать ОТ-эффект №1[I], передав ключи [+][A ↑ ↓ x]
    3. Увеличить у себя[#] статистику «исп. автоабилок[#]» эффекта №0[I] *(0 — текущий)* на 1[I][A ↑ x][+] [A x][+]

В конце хода цели

Нет[+]

Временные модификаторы *(кроме жизни)*

Нет[+]

---

1. ## Дополнительный ОТ-эффект

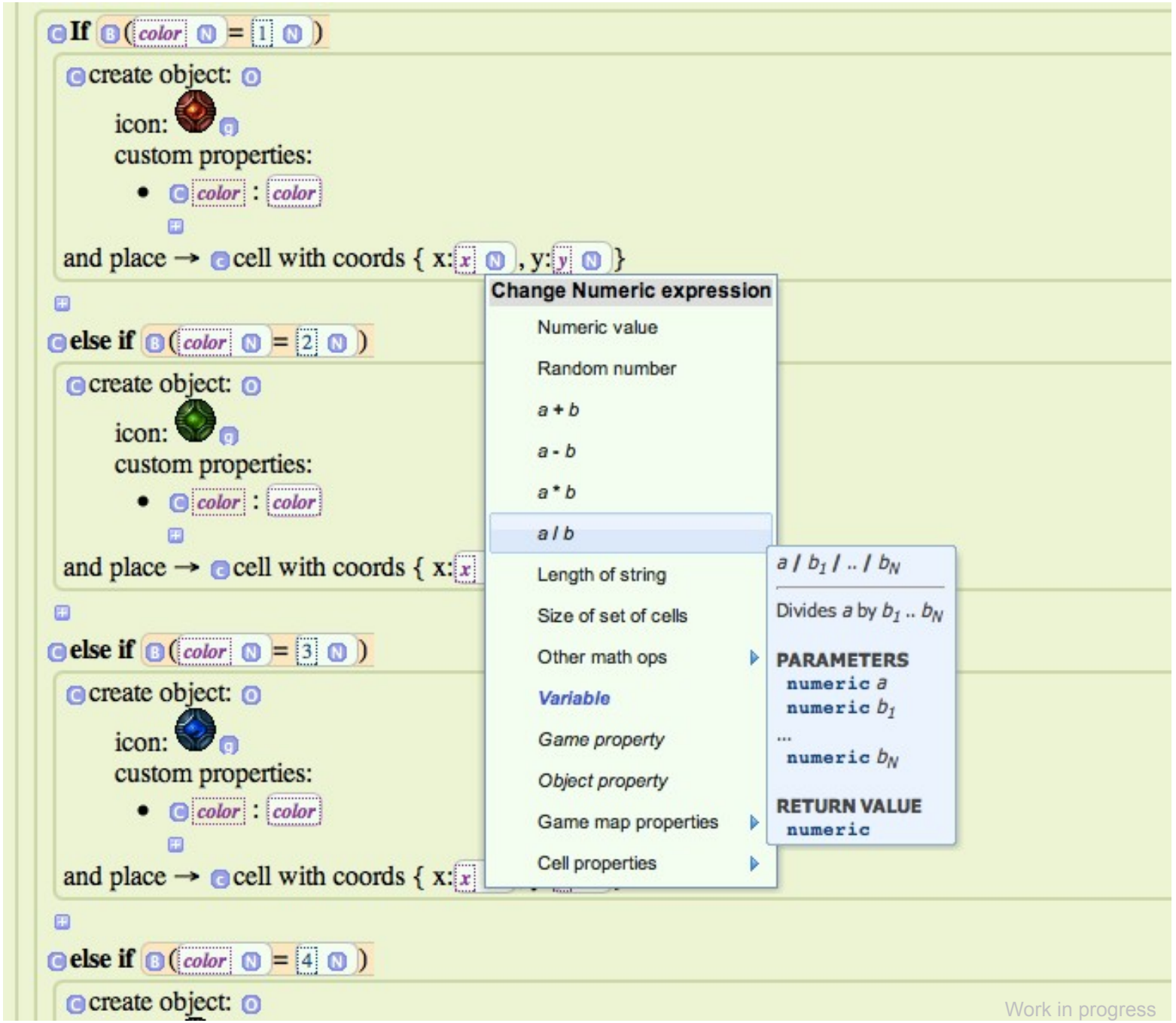
    Цель: на противника[#]
    Время жизни: 5[I] *(≥255 — бессрочно)*
    Период: 0[I]
    Изначальный кулдаун: 0[I]
    Сброс в конце боя: НЕТ[#][B]
    Остается при снятии всех эффектов вручную: ДА[#]

Анализ

- Кто-то предпочитает псевдо-естественный текст, кто-то — блок-схемы.
- Все редакторы принесли существенную пользу.
- Некоторые оказались абсолютно незаменимы.
- В ретроспективе ясно, что вместо работы над некоторыми следовало нанять скрипторов.
- Ни один из редакторов первых шести поколений не оказался достаточно гибким, чтобы выйти за рамки конкретной технологии (движка или даже игры). Но такая задача и не ставилась.

Седьмое поколение

Тулкит для создания визуальных редакторов бизнес-логики


If (color = 1)
create object:
icon:
custom properties:
color : color
and place → cell with coords { x: x , y: y }
else if (color = 2)
create object:
icon:
custom properties:
color : color
and place → cell with coords { x: x
else if (color = 3)
create object:
icon:
custom properties:
color : color
and place → cell with coords { x: x
else if (color = 4)
create object:
Change Numeric expression
Numeric value
Random number
a + b
a - b
a * b
a / b
Length of string
Size of set of cells
Other math ops
Variable
Game property
Object property
Game map properties
Cell properties
a / b1 / .. / bN
Divides a by b1 .. bN
PARAMETERS
numeric a
numeric b1
...
numeric bN
RETURN VALUE
numeric
Work in progress

Цели

- Минимальная стоимость создания новых редакторов.
- Минимальный порог внедрения в «любой» проект на «любой» технологии, даже сторонний.
- Минимальная стоимость поддержки вновь созданных редакторов.
- Достаточная простота освоения и эксплуатации этих редакторов их пользователями.

«No silver bullet»

Даже если вы собираете ваши domain-specific редакторы при помощи идеального сферического тулкита в вакууме, который закрывает любые технические вопросы, вам всё равно *нужно ломать голову* над **юзабилити *каждого* конкретного редактора**!

Природа данных редактора

Редактор работает с *деревом* Control Flow

На основе этого дерева «рендерятся» (генерируются):

- **UI редактора** (у нас — DHTML, псевдоестественный текст).
- **Конечные данные** (код!), с которым работает приложение, для которого сделан редактор.

И то и другое — ***структурированный текст***.

Дерево

Если параметр Мана ≥ 100, то
Нанести урон параметр Мана / 3, цель: оппонент

- Условие
  - Булево выражение
    - «>»
      - Получить значение параметра
        - «Мана»
      - «100»
  - Список действий
    - Действие
      - Нанести урон по цели
        - «/»
          - Получить значение параметра
            - «Мана»
          - «3»
        - «Оппонент»

Конечные данные для нашего примера

Если параметр Мана ≥ 100, то
Нанести урон параметр Мана / 3, цель: оппонент

```
if self:get_param(MANA) > 100 then
    self:deal_damage_to(
            self:get_param(MANA) / 3,
            self:get_opponent()
        )
end
```

Типы нод

Если параметр Мана > 100, то
Нанести урон параметр Мана / 3, цель: оппонент

- Если (Boolean), то (ActionList) → *Root*
- (Action) [, …, (Action)] → *ActionList*
- (Number) > (Number) → *Boolean*
- Параметр (ParamID) → *Number*
- Мана → *ParamID*
- (Пользовательская константа) → *Number*
- Нанести урон (Number), цель: (Target) → *Action*
- (Number) / (Number) → *Number*
- Оппонент → *Target*

«Схема»

- Выделенные нами типы нод (плюс информация о том, какой тип — корневой) определяют *правила конструирования дерева данных редактора*.

- Будем называть *полный набор типов нод* для конкретного редактора логики ***схемой*** (данных) этого редактора.

Схема — фундамент для всех манипуляций с деревом данных

- Валидация дерева данных (проверка на соответствие схеме).
- Дефолтные значения («новый документ» и добавление новых элементов в старый).
- Правила генерации UI и конечных данных.
- Правила работы UI с деревом данных.

Манипуляции с деревом данных в редакторе

- Создать новый документ
- Добавить потомка
- Удалить потомка
- Заменить потомка
- Переставить потомков местами

Допустимость манипуляций

- Допустимость выполнения той или иной манипуляции для каждой ноды напрямую зависит от того, что можно делать с прямыми потомками (первого уровня) ноды данного типа:
  - в списке действий можно поменять элементы местами или изменить их количество;
  - а вот поменять в условии список действий и булево выражение или же совсем удалить булево выражение — уже нельзя.

Классификация типов нод

- Классифицируем типы нод по тому, как ведут себя их прямые потомки.
- Для краткости будем называть «класс типа ноды» «*метатипом*».
- Для ясности, будем вместо «тип ноды» говорить «*конкретный тип ноды*».

Метатипы

Для простоты выберем следующий набор метатипов:

- literal,
- value,
- list,
- record,
- variant.

Если допустить введение «промежуточных» фиктивных нод, при помощи типов, основанных на этом наборе метатипов можно описать практически любую требуемую схему дерева. (Например, list of records of values = dictionary of values, variant of literals = enum.)

Literal

Если параметр Мана ≥ 100, то
Нанести урон параметр Мана / 3, цель: оппонент

- В примере: «оппонент», «Мана».
- Нет потомков.
- Нельзя редактировать.
- В конечных данных — константная строка.

Value

Если параметр Мана ≥ 100, то
Нанести урон параметр Мана / 3, цель: оппонент

- В примере: «100», «3».
- Нет потомков.
- В редакторе пользователь задаёт конкретное значение для каждой ноды.
- В конечные данные попадает значение, заданное пользователем (возможно, предварительно обработанное и в обрамлении какого-то текста).

List

Если параметр Мана ≥ 100, то

Нанести урон параметр Мана / 3, цель: оппонент

- В примере: список действий (с единственным элементом — «нанести урон»).
- «Произвольное» число потомков одного, фиксированного конкретного типа.
- Пользователь может менять число потомков, заменять их, переставлять местами.
- В конечные данные попадают склеенные один за одним значения (с сохранением порядка; возможно, через разделитель и с дополнительным текстом до и после).

Record

Если параметр Мана ≥ 100, то
Нанести урон параметр Мана / 3, цель: оппонент

- В примере: «если..., то», «параметр», «нанести урон …, цель», «>», «/».
- Фиксированное число потомков заданных конкретных типов.
- Конфигурацию прямых потомков нельзя редактировать
- В конечные данные потомки подставляются в заранее определённом порядке (возможно, с каким-то текстом до, после и между ними).

Variant

Если параметр Мана > 100, то
Нанести урон параметр Мана / 3, цель: оппонент

- В примере: булева операция в «если / то», численная операция — величина наносимого урона. (Не были детализованы на слайдах с примерами.)
- Один потомок, конкретный тип которого входит в заранее заданный набор.

Пользователь может редактировать конкретный тип потомка.

- В конечные данные подставляются данные потомка (возможно, в обрамлении какого-то текста).

Генерация конкретных данных и UI редактора

И то и другое — структурированный текст. Для его генерации:

- Обходим дерево данных снизу вверх.
- Получаем конкретный тип и метатип ноды.
- Текст для листовых нод известен сразу.
- Для нелистовых нод подставляем текст потомков в шаблон текста этой ноды.

Текстовые шаблоны для нод

Если параметр Мана ≥ 100, то
Нанести урон параметр Мана / 3, цель: оппонент

Упрощённо, для редактора:

- *Record:* Если ${1}, то
 ${2}
- *List:* Ноды списка действий склеиваем через
- *Record:* ${1} > ${2}
- *Record:* Параметр ${1}
- *Literal:* Мана
- *Value:* ${1}
- *Record:* Нанести урон ${1}, цель: ${2} → *Action*
- *Record:* ${1} / ${2} → *Number*
- *Literal:* Оппонент → *Target*

Создание конкретного редактора

- Описываем схему данных (на Lua-based DSL)
- Из неё генерируем (на Lua):
  - Валидатор данных (на Lua, на JavaScript)
  - Генератор дефолтных данных (на JavaScript)
  - Код редактирования (на JavaScript)
  - Код генерации конкретных данных (на Lua)
- …
- PROFIT!!!

Почему Lua?

Хорошая поддержка «встроенных» декларативных DSL-ей:

```
schema:literal "boolean.true"
{
    editor:ui [[True]]
    {
        "<b>True</b>";
        description = [[<i><b>True</b></i><br>
            <hr>
            The boolean constant <i><b>True</b></i>
          ]];
    };
    render:data
    {
        "true";
    };
}
```

Почему Lua?

Хорошая поддержка Sandboxing'a.

При должной подготовке можно не бояться работать с данными из недоверенных источников.

Если **аккуратно** писать схему, то и получившийся в результате код тоже можно запускать безбоязненно.

Почему Lua?

Мы *любим* этот язык!

За рамками доклада остались

- Схема второго уровня
- Устойчивость данных к изменениям схемы
- Области видимости
- Внешние и внутренние источники данных
- Опциональные потомки
- Подходы к интеграции конечных данных в приложения
- Много-много мелких фичей для тюнинга UI.

Итого

demo.logiceditor.com/**game_ctor**
demo.logiceditor.com/**mini**

Вопросы?

ag@logiceditor.com