Design Patterns in Dynamic Programming

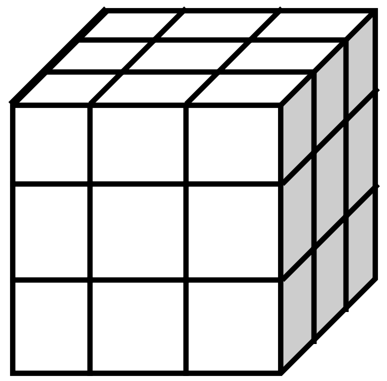
Peter Norvig
Chief Designer, Adaptive Systems
Harlequin Inc.

Outline

- **(1) What Are Design Patterns?**
  Templates that describe design alternatives
- **(2) Design Patterns in Dynamic Languages**
  How to do classic patterns in dynamic languages
  Escape from language limitations
- **(3) New Dynamic Language Patterns**
  New patterns suggested by dynamic languages
- **(4) Design Strategies**
  Thinking about all of software development

(1) What Are Design Patterns?

- **Problem**: Represent a Rubik's Cube as:
  - Cubies[3,3,3] ?
  - Faces[6,3,3] ?
  - Faces[54] ?



- **Design Strategies**:
  - Most important things first (faces, moves)
  - Reuse standard tools (1D), math (permutations)
- **Design Patterns**:
  - Model/View
  - Extension Language (define composed moves)

What Are Design Patterns?

- Descriptions of what experienced designers know (that isn’t written down in the Language Manual)
- Hints/reminders for choosing classes and methods
- Higher-order abstractions for program organization
- To discuss, weigh and record design tradeoffs
- To avoid limitations of implementation language

(Design *Strategies*, on the other hand, are what guide you to certain patterns, and certain implementations. They are more like proverbs than like templates.)

What’s in a Pattern?

- Pattern Name
- Intent / Purpose
- Also Known As / Aliases
- Motivation / Context
- Applicability / Problem
- Solution
- Structure
- Participants
- Collaborations
- Consequences/Constraints
- Implementation
- Sample Code
- Known Uses
- Related Patterns/Compare

From *Design Patterns* and
Pattern Languages of Program Design

Pattern: Abstract Factory

- **Intent**: Create related objects without specifying concrete class at point of creation
- **Motivation**: Portable GUI (Motif, Windows, ...) Create a `ScrollBar`, get a `MotifScrollBar`; Also for `SmallBlueWindow`, `MyAppWindow`
- **Participants**: AbstractFactory, ConcreteFactory, AbstractProduct, ConcreteProduct, Client
- **Sample Code**: `class MotifFactory ... ;`

```
factory = new MotifFactory;
...
CreateWindow(factory, x, y);
```

Level of Implementation of a Pattern

- **Invisible**
  So much a part of language that you don’t notice
  (e.g. when `class` replaced all uses of `struct` in
  C++, no more “Encapsulated Class” pattern)
- **Informal**
  Design pattern in prose; refer to by name, but
  Must be implemented from scratch for each use
- **Formal**
  Implement pattern itself within the language
  Instantiate/call it for each use
  Usually implemented with macros

Sources on Design Patterns

- *Design Patterns*
  Gamma, Helm, Johnson & Vlissides, 1995
- *Pattern Languages of Program Design*
  Coplien & Schmidt, 1995
- *Advanced C++ Programming Styles and Idioms*
  Coplien, 1992
- *Object Models*
  Coad, 1995
- *A Pattern Language*
  Alexander, 1979

(2) Design Patterns in Dynamic Languages

- Dynamic Languages have fewer language limitations
  Less need for bookkeeping objects and classes
  Less need to get around class-restricted design
- Study of the *Design Patterns* book:
  16 of 23 patterns have qualitatively simpler
  implementation in Lisp or Dylan than in C++
  for at least some uses of each pattern
- Dynamic Languages encourage new designs
  We will see some in Part (3)

Design Patterns in Dylan or Lisp

16 of 23 patterns are either invisible or simpler, due to:

- First-class types (6): `Abstract-Factory, Flyweight, Factory-Method, State, Proxy, Chain-Of-Responsibility`
- First-class functions (4): `Command, Strategy, Template-Method, Visitor`
- Macros (2): `Interpreter, Iterator`
- Method Combination (2): `Mediator, Observer`
- Multimethods (1): `Builder`
- Modules (1): `Facade`

First-Class Dynamic Types

- *First-Class*: can be used and operated on where any other value or object can be used
- Types or Classes are objects at run-time (not just at compile-time)
- A variable can have a type as a value
- A type or class can be created/modified at run-time
- There are functions to manipulate types/classes (and expressions to create types without names)
- No need to build extra dynamic objects just to hold types, because the type objects themselves will do

Dynamic Pattern: Abstract Factory

- Types are runtime objects; serve as factories (No need for factory/product dual hierarchy)
- No need for special code; use is invisible:

```
window-type := <motif-window>;
...
make(window-type, x, y);
```

- Still might want factory-like objects to bundle classes (window, scroll-bar, menu, border, tool-bar, ...)
- Works in Lisp or Dylan or Smalltalk or ...
- Dylan classes explicitly `abstract` or `concrete`

Pattern: Abstract Factory

- Static version requires dual hierarchy of classes:

  ```
  GUIFactory                 Window
  NTFactory                  NTWindow
  MacOSFactory               MacOSWindow
  XFactory                   XWindow
  MotifFactory               MotifWindow
  ```

  with objects instantiated on both sides
- Dynamic version needs only the `Window` classes
  The classes themselves serve as factories
  This works because classes are first-class values
  We can say `make(c)`

First-Class Dynamic Functions

- Functions are objects too
- Functions are composed of methods
- There are operations on functions (compose, conjoin)
- Code is organized around functions as well as classes
- Function closures capture local state variables (*Objects* are state data with attached behavior; *Closures* are behaviors with attached state data and without the overhead of classes.)

Pattern: Strategy

- **Intent**: Define a family of interchangeable algorithms
- **Motivation**: Different line-breaking algorithms
- **Participants**: Strategy, ConcreteStrategy, Context
- **Implementation**:

```
class Compositor ...;
class TeXCompositor : public Compositor...;
class Composition {
  public: Composition(Compositor*); ...};
...
Composition* c =
  new Composition(new TeXCompositor);
c.compositor->Compose();
```

Dynamic Pattern: Strategy

- The strategy is a variable whose value is a function (E.g., with first-class functions, pattern is invisible)
- **Implementation**:
```
compositor := TeXcompositor;
compositor(...);
```
- General principle: no need for separate classes that differ in one (or a few) well-understood ways.
- May still want strategy objects:
```
make(<strategy>, fn: f, cost: 5, speed: 4)
```
but don't need separate classes for each instance

Macros

- Macros provide syntactic abstraction
  You build the language you want to program in
- Just as important as data or function abstraction
- Languages for Macros
  - String substitution (cpp)
  - Expression substitution (Dylan, extend-syntax)
  - Expression computation (Lisp)
    Provides the full power of the language while you are writing code

Pattern: Interpreter

- **Intent**: Given a language, interpret sentences
- **Participants**: Expressions, Context, Client
- **Implementation**: A class for each expression type
  An `Interpret` method on each class
  A class and object to store the global state (context)
- No support for the parsing process
  (Assumes strings have been parsed into exp trees)

Pattern: Interpreter with Macros

- **Example**: Definite Clause Grammars
- A language for writing parsers/interpreters
- Macros make it look like (almost) standard BNF
  ```
  Command(move(D)) -> “go”, Direction(D).
  ```
- Built-in to Prolog; easy to implement in Dylan, Lisp
- Does parsing as well as interpretation
- Builds tree structure only as needed
  (Or, can automatically build complete trees)
- May or may not use expression classes

Method Combination

- Build a method from components in different classes
- **Primary** methods: the “normal” methods; choose the most specific one
- **Before/After** methods: guaranteed to run;
  No possibility of forgetting to call `super`
  Can be used to implement *Active Value* pattern
- **Around** methods: wrap around everything;
  Used to add tracing information, etc.
- Is added complexity worth it?
  Common Lisp: Yes;   Most languages: No

Pattern: Observer

- **Intent**: When an object changes, notify all interested
- **Motivation**: A spreadsheet and a bar chart are both displaying the results of some process. Update both displays when the process gets new numbers.
- **Participants**: Subject, Observer, ConcreteSubject, ConcreteObserver
- **Implementation**:
  Subject: methods for attach/detach observer, notify
  Observer: method for update

Observer with Method Combination

- *Observer* is just “notify after every change”
  (With more communication in complex cases)
- **Implementation**: Use `:after` methods
  Can be turned on/off dynamically if needed
  Allows the implementation to be localized:

```
(mapc #’notify-after ‘(cut paste edit ...))
(defun notify-after (fn)
  (eval `(defmethod ,fn :after (x)
          (mapc #‘notify (observers x)))))
```

- Note no implementation needed in Subject class
- See *Relation* pattern for `observers` implementation

The Type/Operation Matrix

- Programs have types and operations:

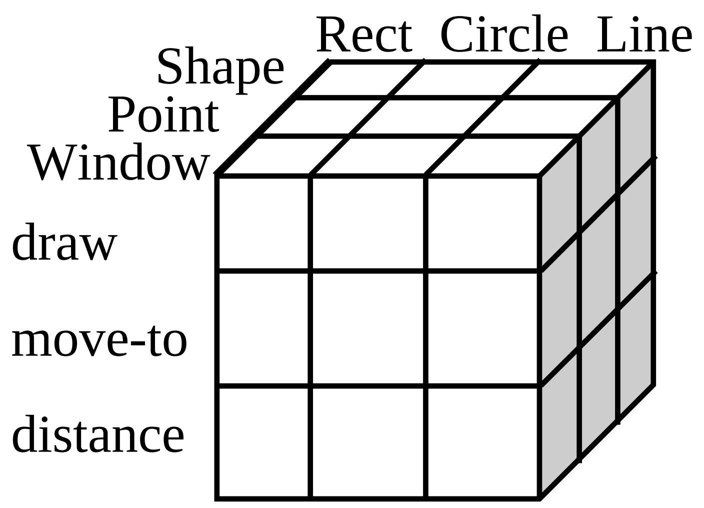
	Rectangle	Circle	Line
draw			
position			
area			

Three types of programming fill cells in different order:

- **Procedural**: write entire row at a time
  (Problems with case statements)
- **Class-Oriented**: write column at a time (inherit some)
- **Literate**: fill cells in any order for best exposition

Multimethods

- Operations often deal with multiple objects: `f(x,y)`



- Class-oriented has a distinguished object: `x.f(y)`
  (May be unnatural, hard to extend)
- Multimethods allow literate programming
- Support *Singleton* and prototypes using == dispatch

Pattern: Builder

- **Intent**: Separate construction of complex object from its representation; so create different representations
- **Participants**: Builder, ConcreteBuilder, Director, Product
- **Motivation**: Read text document in RTF format
  - Convert to one of many formats
  - One conversion algorithm
  - Details differ depending on target format
- **Implementation**: Separate class for each type of object to build; another for the “director”

Pattern: Builder

- **Builder**: TextConverter class with methods for ConvertCharacter, ConvertParagraph, ...
- **ConcreteBuilder**: ASCIIConverter, TeXConverter, ...
- **Director**: Builder slot and algorithm for conversion
- **Product**: ASCIIText, TeXText, ...
- Total of $2n + 2$ classes
- **Implementation**:

```
switch(t=GetToken().Type) {
  CHAR: builder->ConvertChar(t);
  FONT: builder->ConvertFont(t);
  PARA: builder->ConvertParagraph(t);}
```

Pattern: Builder with Multimethods

- No builder or director classes; *n* product classes
- One builder function (extensible: no `switch`)
- *n* methods for conversion (`convert`)
- **Implementation**:

```
target-class := <TeX-Text>;
target := make(target-class);
...
token := get-token();
convert(token, token.type, target);
...
define method convert
 (token, type==#”font”, target::<TeX-Text>)
```

Modules

- In C++, classes organize, implement object behavior *and* define name spaces
- This leads to problems:
  - Compromises between two purposes
  - Need more selective access than public/private
  - Friend classes don’t work well
- Separate modules relieve the class of double-duty
- Can have multiple modules for one library of code

Pattern: Facade

- **Intent**: Provide a simple interface to a subsystem
- **Motivation**: A complex system may have many pieces that need to be exposed. But this is confusing. Supply a simpler interface on top of the system.
- **Participants**: Facade, SubsystemClasses
- **Example**: A Compiler class that calls scanner, parser, code generator in the right way
- **Facade** pattern with **modules** is invisible
  - Don't need any bookkeeping objects or classes
  - Just export the names that make up the interface

Other Invisible Patterns

- The following patterns are invisible in dynamic languages, and usually implemented more efficiently
- **Smart Pointers**
  (Pointers that manage copy constructors)
- **Reference Counting**
  (Automatic memory management)
- **Closures**
  (Functions with bound variables)
- **Wrapper Objects**
  (Objects with one data member, a primitive type such as a character or 32-bit integer)

(3) New Dynamic Language Patterns

- First-Class Patterns: make the design more explicit
- Iterators: a study of C++, Dylan, Smalltalk and Sather
- Mixing compile time and run time
  (Memoization, Compiler, Run time loading,
  Partial Evaluation)
- Freedom of syntactic expression
  (Decision tables, Rule-based translator)
- Freedom from implementation details
  (Relation)

First-Class Design Patterns

- Define the pattern with code, not prose
- Use the pattern with function or macro call(s), not a comment
- Implement with classes, objects, functions, macros
- This is the second half of abstraction:
  Assigning *something* to a name.
  It works better when *something* is a real object.
  (It is hard because many patterns are not localized.)
- It's easier when code needn't be organized by class
  Then the call to the pattern can generate any code

First Class Pattern: Subroutine

- Long ago, *subroutine call* was just a pattern
- Involves two parts: call and definition

```
load R1, x            SQRT:
load R0, *+2          ...
branch SQRT           branch @R0
```

- Nowadays, made formal by the language

```
sqrt(x);              function sqrt(x) ...
```

- Note there are still 2 parts in formal use of pattern
- Many patterns are harder to define formally because their use is spread out over more than two places

First Class Pattern Implementation

- As **abstract class**:

```
define class <adapter> ()
  slot adaptee;
end
```

- As **generic function:**

```
define generic iteration-protocol(object)
```

- As a **macro**:

```
define grammar
  Command(go(D)) -> “go”, Direction(D);
  ...
end;
```

Pattern: Protocol Method

- **Intent**: Implement set of related operations
- **Implementation**: Define a `protocol` method that returns the required functions. Arrange to call the functions as needed.
- **Participants**: Protocol generic function, Client(s)
- **Example**: Protocol returns 2 objects, 3 functions:

```
iteration-protocol(object) =>
   state, limit, next, done?, current
```

- **Advantages**: Doesn't require unique parent class
  Can be quicker to compute all at once
  Often avoid allocating bookkeeping objects, classes

Pattern: Protocol Method

- Interfaces have 3 potential users, those who want to:
  - Use existing code properly
  - Extend an existing class
  - Implement for a brand new base class
- Protocols can make this distinction
- Classes can also make it, via virtual functions
  (But don’t allow a new class not derived from base)

A Study in Patterns: Iterator

- **Intent**: allow access to each element of collection
- **Motivation**: separate interface/implementation, allow multiple accesses to same collection
- **Participants**: Iterator, ConcreteIterator, Collection, ConcreteCollection
- **C++ Implementation**: Problems: Creating, deleting iterators; Need for dual hierarchy; Ugly syntax:

```
ListIter<Employee*>* i=employees->Iter();
for (i.First(); !i.IsDone(); i.Next());
  i.CurrentItem()->Print();
delete i;
```

C++ Pattern: Internal Iterator

- **Intent:** An iterator to which you provide an operation that will be applied to each element of collection
- **Example**: print a list of employees

```
template <class Item> class List
template <class Item> class ListIter
  public: bool Traverse();
  protected: virtual bool Do(Item&);
class PrintNames : ListIter<Employee*>
  protected: bool Do(Employee* & e) {
    e->Print();}
...
PrintNames p(employees); p.Traverse();
```

Smalltalk Pattern: Internal Iterator

- Closures eliminate the need for iterator classes
  (Replace 10 or so lines of code with 1)
- Pass a block (function of one arg) to the `do:` method
  ```
  employees do: [ :x | x print ]
  ```
- Easy for single iteration
- Also used heavily in Lisp, Dylan
- Inconvenient for iteration over multiple collections
  How do you compare two collections?
  How do you do element-wise A := B + C?

Dylan Pattern: Iteration Protocol

- Iteration *protocol* instead of iterator classes
- The protocol returns 2 objects, 3 functions:
```
iteration-protocol(object) =>
  state, limit, next, done?, current
```
- Designed for optimization (see Lazy Evaluation)
- No need for parallel class hierarchy of iterators
  Do need to provide (or inherit) iteration protocol
- Capability to define operations on protocol results
  More flexible algebra of iterators
  (`reverse`, `first-n`, `lazy-map`)

Dylan Pattern: Iteration Protocol

- Simple syntax

```
for(i in collection) print(i) end;
```

- Multiple iteration allowed with more complex syntax

```
for(i in keys(A), x in B, y in C)
  A[i] := x + y;
end;
```

- Dylan also supports internal iteration:

```
do(print, collection)
```

- Many internal iterators (higher-order functions):

```
always?(\=, A, B);
map-into(A, \+, B, C);
```

Dylan: Iteration Protocol Algebra

- Add a class named `<iterator>` with slots `object` and `protocol` such that:

```
iteration-protocol(i :: <iterator>) =>
  protocol(i.object)
```

- Add functions to make objects of this class:

```
define function backward (collection)
  make(<iterator>, object: collection,
    protocol: reverse-iteration-protocol);
```

- Use the functions to build `<iterator>`s:

```
 for (x in C.backward) ... end;
```

- This may soon be built-in to Dylan’s syntax:

```
 for (x in C using reverse-iteration-protocol)
```

Pattern: Lazy Mapper Iteration

- Adding a lazy mapper iterator

```
make(<iterator>,object: c, protocol: f.lazy-mapper)
```

- Implementing the lazy-mapper:

```
define function lazy-mapper (fn)
 method (coll)
  let (state, lim, next, done?, current) =
    iteration-protocol(coll);
  let mapper = method (c, state)
                 fn(current(c, state));
               end;
  values(state, lim, next, done?, mapper)
 end;
end;
```

Sather Pattern: Coroutine Iterator

- Notion of iterators as coroutines. In ARRAY class:

```
index!:INT is
  loop yield 0.to!(self.size-1) end
end;
elt!:T is
  loop yield self[self.index!] end
end;
```

- Anonymous iteration: no need for variable names:

```
loop
  A[A.index!] := B.elt! + C.elt!
end;
```

Pattern: Coroutine

- **Intent**: separate out distinct kinds of processing; save state easily from one iteration to the next
- **Implementation**: Most modern language implementations support an interface to the OS's threads package. But that has drawbacks:
  - No convenient syntax (e.g. `yield`, `quit`)
  - May be too much overhead in switching
  - Problems with locking threads
- **Implementation**: Controlled uses of coroutines can be compiled out (Sather iters, Scheme call/cc)

Pattern: Control Abstraction

- Most algorithms are characterized as one or more of:
  Searching: (find, some, mismatch)
  Sorting: (sort, merge, remove-duplicates)
  Filtering: (remove, mapcan)
  Mapping: (map, mapcar, mapc)
  Combining: (reduce, mapcan, union, intersection)
  Counting: (count)
- Code that uses these higher-order functions instead of loops is concise, self-documenting, understandable, reusable, usually efficient (via inlining)
- Inventing new control abstractions is a powerful idea

Pattern: New Control Abstraction

- **Intent**: Replace loops with named function or macro
- **Motivation**: A control abstraction to find the best value of a function over a domain, `find-best`
- **Examples**:

```
find-best(score, players);
find-best(distance(x), numbers, test: \<);
```

  where `define function distance(x)`

```
       method (y) abs(x - y) end; end;
```

- **Implementation**: A simple loop over the collection, keeping track of best element and its value.
  In some cases, a macro makes code easier to read

Pattern: Memoization

- **Intent**: Cache result after computing it, transparently
- **Example**:
```
(defun-memo simplify (x) ...)
```
- **Implementation:** Expands into (roughly):
```
(let ((table (make-hash-table))
  (defun simplify (x)
    (or (gethash x table)
        (setf (gethash x table) ...))))
```
- **Complications**: Know when to empty table, how many entries to cache, when they are invalid

Pattern: Singleton as Memoization

- Can use memoization to implement Singleton pattern
- **Implementation**:
```
(defmethod-memo make ((class SINGLETON))
  ...)
```
- **Invisible Implementation**: Don’t need singletons if you can dispatch on constants:
```
define constant s1 = make(<class>, n: 1);
define method m (x == s1) ... end

define constant s2 = make(<class>, n: 2);
define method m (x == s2) ... end
```

Pattern: Compiler

- Like the *Interpreter* pattern, but without the overhead
- A problem-specific language is translated into the host programming language, and compiled as normal
- Requires complex Macro capabilities
  May or may not require compiler at run time
- A major factor when Lisp is faster than C++
- In a sense, every macro definition is a use of the *Compiler* pattern (though most are trivial uses)
- **Examples**: Decision trees; Window, menu layout; Definite Clause Grammar; Rule-Based Translator

Pattern: Run-Time Loading

- **Intent**: Allow program to be updated while it is running by loading new classes/methods (either patches or extensions). Good for programs that cannot be brought down for upgrades.
- **Alternative Intent**: Keep working set small, start-up time fast by only loading features as needed
- **Implementation**: DLLs, dynamic shared libraries. Language must allow redefinition or extension

Pattern: Partial Evaluation

- **Intent**: Write literate code, compile to efficient code
- **Example**:

```
define function eval-polynomial(x, coefs)
  let sum = 0;
  for (i from 0, c in coefs)
    sum := sum + c * x ^ i;
  end;
  sum;
end;
```

such that `eval-polynomial(x, #[1, 2, 3])`
compiles to `0 + 1 + 2 * x + 3 * x * x`
or better yet `1 + x * (2 + 3 * x)`

Pattern: Partial Evaluation

- **Implementation**: Mostly, at whim of compiler writer (Harlequin Dylan, CMU Lisp compilers good at it)
- **Alternative Implementation**: Define a problem-specific sublanguage, write a compiler for it with partial evaluation semantics
- **Example**:
  Macro call `horner(1 + 2 * x + 3 * x ^ 2)`
  expands to `1 + x * (2 + 3 * x)`

Pattern: Rule-Based Translator

- **Intent**: For each pattern detected in input, apply a translation rule
- Special case of *Interpreter* or *Compiler*
- **Example**:

```
define rule-based-translator simplify ()
  (x + 0) => x;
  (x * 1) => x;
  (x + x) => 2 * x;
  (x - x) => 0;
  ...
end;
```

Pattern: Relation

- **Intent**: Represent that *x* is related to *y* by *R*
- **Motivation**: Often, this is done by making a *R* slot in the class of *x* and filling it with *y*. Problems:
  - May be no common superclass for *x*'s
  - *y* may take less than a word (say, 1 bit)
  - Don't want to waste space if most *y*'s are void
  - Don't want to page if cycling over *R*'s
- **Solution**: Consider a range of implementations, from slot to bit vector to table to data base. Provide a common interface to the implementations.

(4) Design Strategies

- **What to Build**
  (Class libraries, frameworks, metaphors, ...)
- **How to Build**
  (Programming *in*, *into*, and *on* a language)
- **How to Write**
  (Literate programming vs. class-oriented/obsessed)
- **Specific Design Strategies**
  (Open Implementation; English Translation)
- **Metaphors: The Agent Metaphor**
  (Is *agent-oriented programming* the next big thing?)
- **Combining Agent Components**

What to Build

- **Class Libraries / Toolkits**
  Generic (sets, lists, tables, matrices, I/O streams)
- **Frameworks**
  Specialized (graphics), “Inside-Out” (callbacks)
- **Languages**
  Generic or Specialized (Stratified Design)
- **Design Process**
  Source control, QA, Design rationale capture, ...
- **Metaphors**
  Agent-Oriented, Market-Oriented, Anytime Programming

How to Build

- **Programming *In* a language**
  The design is constrained by what the language offers
- **Programming *Into* a language**
  The design is done independently of language, then the design is implemented using features at hand
- **Programming *On* a language**
  The design and language meet half way. This is programming *into* the language you wish you had; a language you build *on* the base language.
  Sometimes called *Stratified Design.*

How to Build: Abstraction

- Data abstraction: encapsulation, first-class types
- Functional abstraction: first-class functions, closures
- Syntactic abstraction: macros, overloading
- Control abstraction: macros and high-order functions
- Design process abstraction: abstract away files, deal with phases of project, explicit development process
- Resource abstraction: separate what it takes to do it from what is done (See *Open Implementation*)
- Storage abstraction: garbage collection, no `new`, slot access and function calls have same syntax

How to Write: Literate Programming

- Literate Programming: allow programmer to decide how best (in what order) to present the program
- Obsession: insisting on one’s favorite organization
- Class-Oriented Prog: Organize text around classes
- Class-Obsessed Prog: Doing this to an extreme
- C++: Oriented to class and copy, *not* pure objects
- Lisp, Dylan: Oriented to pure objects, modules, literate programming, *not* class over functions
- Anti-Object-Obsessed: *“I do not believe in things. I believe only in their relationships”* - George Braque

Class-Oriented or Class-Obsessed?

- Class-based textual organization good for elementary abstract data types
- Good to have some organization guidelines
- C++ provides several escapes from class-obsession
- C++ encourages bookkeeping classes (*Visitor* pattern serves only to get around restriction)
- Need bookkeeping especially for *n*-ary relations
- `friend` and related accesses are complex
- Class-based names don't replace a real module system
- Class-oriented organization prevents certain macros

Strategy: Open Implementation

- **Intent**: Open up the black box; performance counts
- **Motivation**: A spreadsheet could be implemented by making 100x100 small windows. The window system's interface allows this, but it would be inefficient. Could we persuade the system to use an efficient implementation just this once? Then we don't have to re-code all the stuff that already works.
- **Idea**: Complex interfaces are split in two: one for the specification, and one for the implementation. When it matters, specify the implementation you need
- (See *Programmable Programming Language*)

Design Strategy: English Translation

- To insure that your program says what you mean:
  (1) Start with English description
  (2) Write code from description
  (3) Translate code back to English; compare to (1)
- **Example:** (1), (2) from a Lisp textbook
  (1) “Given a list of monsters, determine the number
  that are swarms.”
  (2) *See next slide*
  (3) “Given a list of monsters, produce a 1 for a
  monster whose type is swarm, and a 0 for others.
  Then add up the numbers.”

Design Strategy: English Translation

- Example, step (2):

```
(defun count-swarms (monsters)
  (apply ‘+ (mapcar
             #’(lambda (monster)
                  (if (eql (type-of monster)
                           ‘swarm)
                      1 0))
             monsters)))
```

- (Small changes not relevant to problem were made)

Design Strategy: English Translation

- Code taking the strategy into account:
- (1) “Given a list of monsters, determine the number that are swarms.”
- (2) A straight-forward implementation:

```
(defun count-swarms (monsters)
  (count ‘swarm monsters :key #’type-of))
```

- (3) “Given a list of monsters, count the number whose type is swarm.”

Metaphor: Agent Programming

Traditional Program	**Agent Program**
Function	Agent
Input / output	Percept / action
Logic-based	Probability-based
Goal-based	Utility-based
Sequential, single-	Parallel, multi-
Hand Programmed	Trained (Learning)
Design trade-offs	Run-time trade-offs
Fidelity to expert	Perform well in env.

Agent Programming Technology

Mathematics

- Decision Theory
- Control Theory
- Statistical Optimization
- Economic Theory
- Markov Decision Processes

Artificial Intelligence

- Machine Learning
- Neural Networks
- Reinforcement Learning
- Bayesian Networks
- Anytime Programming

Design for a Rational Agent

- Calculate $P(\textit{current state})$
  - Based on evidence, percept, last action
- Calculate $P(Result(Act))$,$U(Result(Act))$
  - Nondeterministic: many states, results
- Calculate expected utility *EU* for each action
  - $EU(Act) = \Sigma_i \; P(Result_i(Act)) \cdot U(Result_i(Act))$
- Choose the Action with highest expected utility
  - $Best\ Act = \mathrm{argmax}_A \; EU(Act_A)$
- **Approximate** if not enough resources to compute

Rational Reasoning

- Obey Principle of Maximum Expected Utility
- Apply at design or run time as appropriate
- Not a new idea: *“To judge what one must do to obtain a good or avoid an evil, it is necessary to consider not only the good and the evil in itself, but also the probability that it happens or does not happen; and to view geometrically the proportion that all these things have together.”*
- A. Arnauld, *The Art of Thinking*, 1662
- Has been the basis of most science since then (Economics, Medicine, Genetics, Biology, OR, ...)

The Three Laws of Robotics

- (1) Don’t harm humans, through action or inaction
- (2) Obey humans, except when conflict with (1)
- (3) Protect self, except when conflict with (1, 2)
- Why Asimov was wrong
  - Too Boolean: need notions of utility, probability
  - Problems with “cause,” “protect,” “harm,” etc.
- Laws can be seen as defining utility function only
  - Still too absolute
- Actually, Asimov probably knew it (*Roundabout*)

Object-Oriented Programming

Lessons Learned:

- Abstraction: data (objects), procedural (interfaces)
  - *What*, not *how*, it computes
- No global variables; no top level
  - Any computation might be embedded
- Reuse through inherit and modify
- Composition through standard techniques:
  - Conditional, sequential, loop/recursion
  - *P* is closed under composition
    (But real programmers make finer distinctions)

Agent Programming

Lessons Learned:

- Plan abstraction
  - *What*, not *how*, it acts
  - Resource allocation optimized separately (MS)
- No top level goals
  - Any agent can be retargetted
- Reuse through parameter-setting optimization
- Composition is not straightforward:
  - Economic (Market-Oriented) Programming
  - Anytime Programming

Combining Agent Components

- Essential for modular, scaleable, reusable systems
- Reuse in new or changed environment
  - Machine learning/statistical optimization
- Reuse with retargeted goal or utility function
  - Real advantage over traditional programming
- Allocating resources to agent components/tasks
  - Anytime programming
- Scaling up to multiple cooperating agents
  - Economic (Market-Oriented) Programming

Real-Time Resource Allocation

- Sensing and planning as information sources
  - Manage based on value of information
  - Assumes time-dependent utility function
  - Value depends on quality, time, ease of use
- Trade-off value of information vs. resources
  - Build out of anytime and contract components (Interrupt when results are good enough)
  - Modularize construction vs. optimization
  - Maintain conditional performance profiles

Compilation of Anytime Algorithms

- Given: components with performance profiles, *Q*
  - *Interpret(Data);*   *Q(Interpret, t)* = ...
  - *PlanPath(A, B, S);* *Q(PlanPath, Qs, t)* = ...
- Given: an abstract overall algorithm
  - E.g. *A* = *PlanPath(A, B, Interpret(Camera()))*
  - $Q(A,t) = \max Q(PlanPath, Q(Interpret, t_1), t_2)$
    where $t = t_1 + t_2$
- Find optimal allocation of resources
  - Monitor and adapt at run-time

Technology for Multi-Agent Systems

- Market-Oriented Programming
  - Bid in a competitive market of resources
  - The market optimizes the value of resources
- Protocol Engineering
  - Make the market communication efficient
- Incentive Engineering
  - Achieve good for community
- Natural Language (and other) Communication
  - Communication among programs and humans