Introducing the core concepts of programming with DJ

- Define graphs, paths and strategies.
- The meaning of a strategy is a path set.
- Path sets may be infinite but are represented efficiently by traversal graphs.
- Traversal graph construction is provided by DJ and is covered by a US patent.
- Traversal graph construction may be covered later but it is unimportant as long as you understand the meaning of a strategy to be certain path set.

Graphs and paths

- Directed graph: *(V,E)*, *V* is a set of nodes, *E Í V´ V* is a set of edges.
- Directed labeled graph: *(V,E,L)*, *V* is a set of nodes, *L* is a set of labels, *E Í V´ L´ V* is a set of edges.
- If *e* = *(u,l,v)*, *u* is source of *e*, *l* is the label of *e* and *v* is the target of *e*.

Graphs and paths

- Given a directed labeled graph: *(V,E,L)*, a node-path is a sequence $p = <v_0 v_1 \ldots v_n>$ where $v_i \in V$ and $(v_{i-1}, l_i, v_i) \in E$ for some $l_i \in L$.
- A path is a sequence $<v_0\ l_1\ v_1\ l_2 \ldots l_n\ v_n>$, where $<v_0 \ldots v_n>$ is a node-path and $(v_{i-1}, l_i, v_i) \in E$.

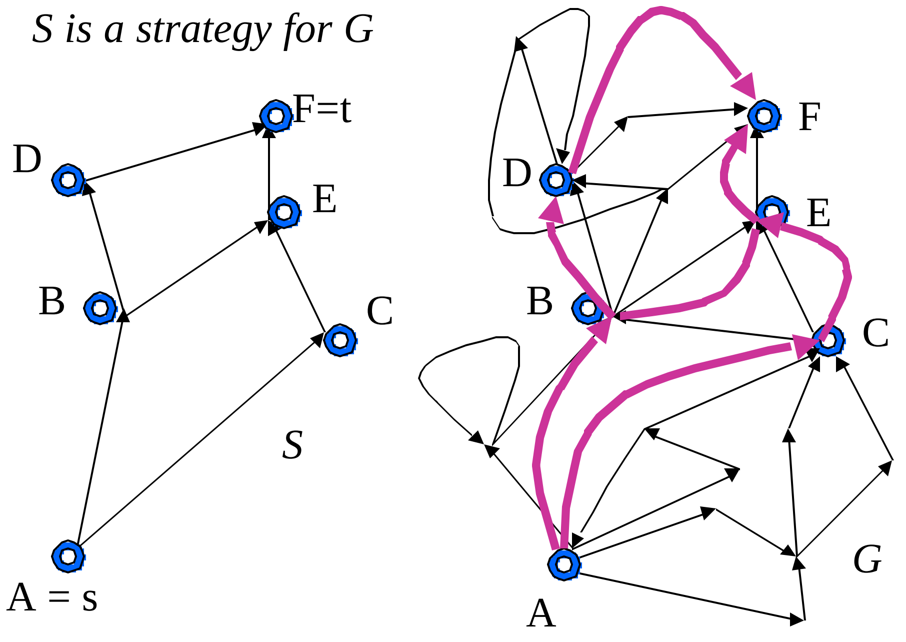
Graphs and paths

- In addition, we allow node-paths and paths of the form $<v_0>$ (called trivial).
- First node of a path or node-path *p* is called the source of *p*, and the last node is called the target of *p*, denoted *Source(p)* and *Target(p)*, respectively. Other nodes: interior.

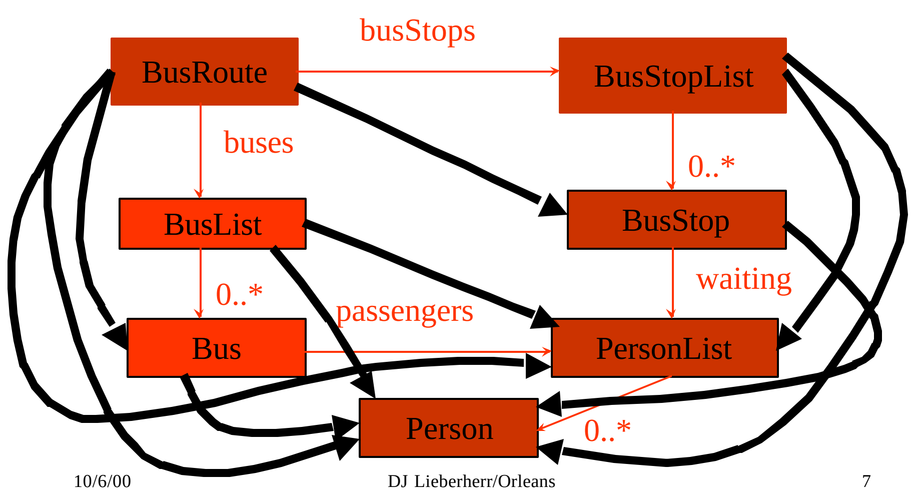
Strategy definition: embedded, positive strategies

- Given a graph G, a strategy graph S of G is any subgraph of the transitive closure of G.
- The transitive closure of G=(V,E) is the graph G*=(V,E*), where E*={(v,w): there is a path from vertex v to vertex w in G}.

S is a strategy for G



Transitive Closure



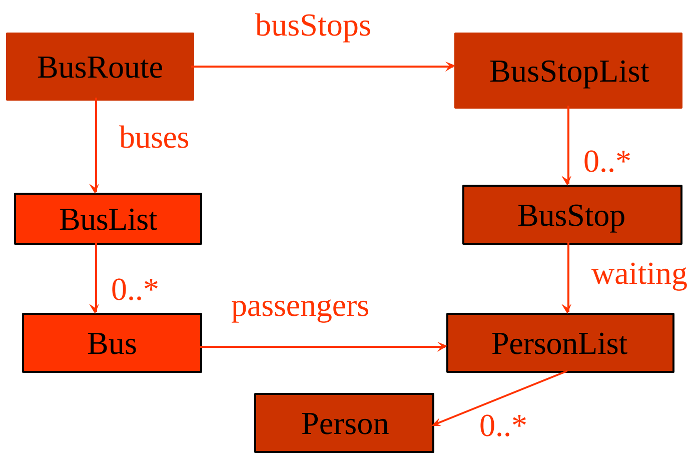
Strategy graph and base graph are directed graphs

Key concepts

- Strategy graph *S* with source *s* and target *t* of a base graph *G*. *Nodes(S)* subset *Nodes(G)* (Embedded strategy graph).
- A path *p* is an ***expansion*** of path *p'* if *p'* can be obtained by deleting some elements from *p*.
- *S* defines ***path set*** in *G* as follows: *PathSet*$_{st}$*(G,S)* is the set of all *s*-*t* paths in *G* that are expansions of any *s*-*t* path in *S*.

Expansion

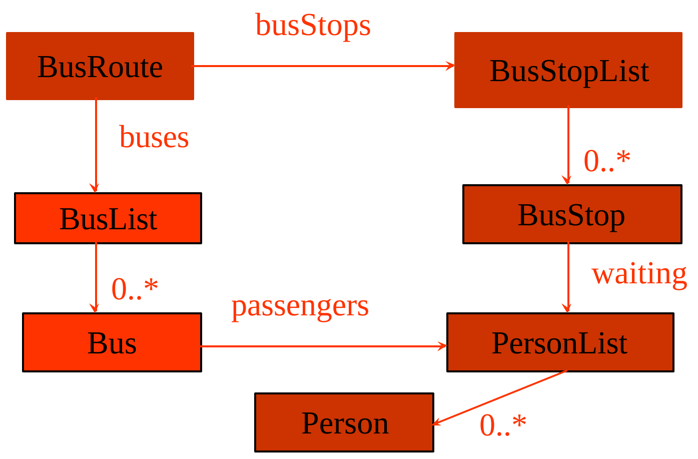
(BR BSL BS PL P) is an expansion of (BR BS P)



`S = from` BusRoute `to` Person

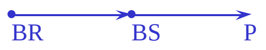
PathSet

$\text{PathSet}_{\text{BusRoute,Person}}(S,G)=\{(\text{BR BL B PL P}),(\text{BR BSL BS PL P})\}$

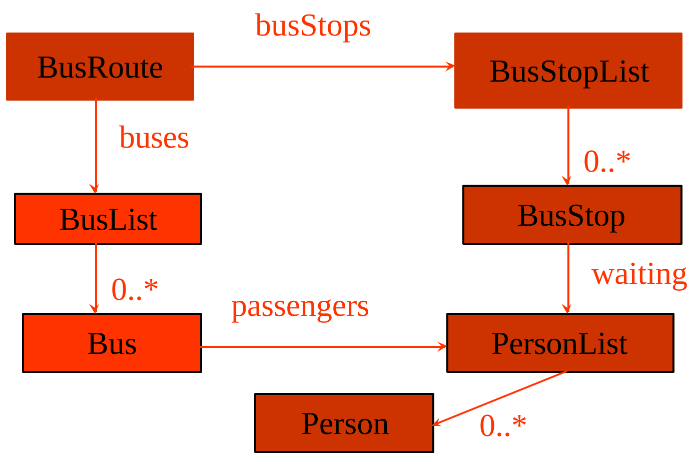


`S = from` BusRoute `through` BusStop `to` Person

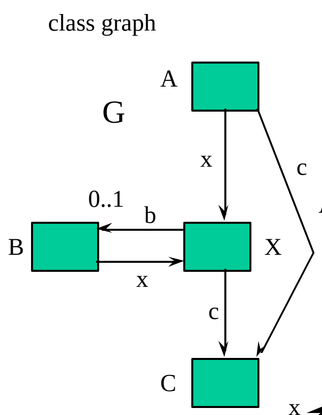
PathSet



$\text{PathSet}_{\text{BusRoute,Person}}(S,G)=\{(\text{BR BSL BS PL P})\}$

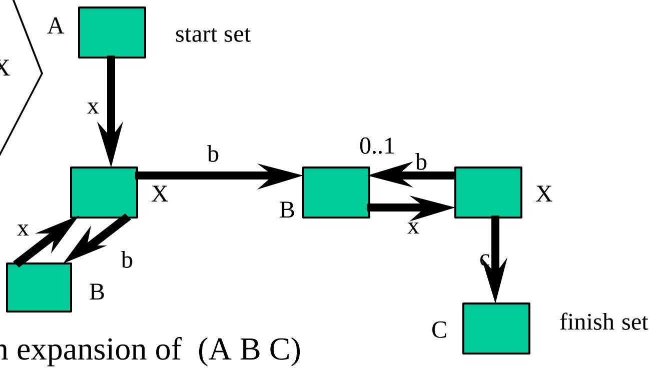


Expansion PathSet



```
strategy:
{A -> B
 B -> C}
```
S

Traversal graph



(A X B X C) is an expansion of (A B C)

$\text{PathSet}_{A,C}(S,G) = \{(A\ X\ B\ X\ (B\ X)^*\ C)\}$

= defined by traversal graph

DJ

- An implementation of AP using only the DJ library (and the Java Collections Framework)
- All programs written in pure Java
- Intended as prototyping tool: makes heavy use of introspection in Java
- Integrates Generic Programming (a la C++ STL) and Adaptive programming

Integration of Generic and Adaptive Programming

- A traversal specification turns an object graph into a list.
- Can invoke generic algorithms on those lists. Examples of generic algorithms: add, remove, contains, etc.
- What is gained: genericity not only with respect to data structure implementations but also with respect to class graph.

Sample DJ code

```
// Iterate through library users
List libUsers =
  classGraph.asList(library,
     "from Library to User");
ListIterator li =
  libUsers.listIterator();
// iterate through libUsers
```

Methods provided by DJ

- On ClassGraph, ObjectGraph, TraversalGraph, ObjectGraphSlice: `traverse, fetch, gather`
- traverse is the important method; fetch and gather are special cases
- TraversalGraph
  – Object traverse(Object o, Visitor v)
  – Object traverse(Object o, Visitor[] v)

Traverse method: excellent support for Visitor Pattern

```
// class ClassGraph
Object traverse(Object o,
  Strategy s, Visitor v);
```

traverse navigates through Object o following traversal specification s and executing the before and after methods in visitor v

ClassGraph is computed using introspection

Fetch Method

- If you love the Law of Demeter, use fetch as your shovel for digging:
  - Part k1 = (K) classGraph.fetch(a,"from A to K");
- The alternative is (digging by hand):
  - Part k1 = a.b().c().d().e().f().g().h().i().k();
- DJ will tell you if there are multiple paths to the target (but currently only at run-time).

Gather Method

- Returns a list of copied objects.
- Object ClassGraph.gather(Object o, String s)
  - List ks = classGraph.gather(a,”from A to K”); returns a list of K-objects.

Using DJ

- traverse(…) returns the v[0] return value. Make sure the casting is done right, otherwise you get a run-time error. If “`public Object` getReturnValue()” returns an Integer and traverse(…) casts it to a Real: casting error at run-time.
- Make sure all entries of Visitor[] array are non-null.

Using multiple visitors

```
// establish visitor communication
aV.set_cV(cV);
aV.set_sV(sV);
rV.set_aV(aV);

Float res = (Float) whereToGo.
   traverse(this,
   new Visitor[] {rV, sV, cV, aV});
```

DJ binaryconstruction operations

	cg	s	tg	o	og	ogs
cg	*	tg,cg	*	og	*	*
s		*	*	*	ogs	*
tg			*	*	ogs	*
o				*	*	*
og					*	*
ogs						*

Who has traverse, fetch, gather? (number of arguments of traverse)

	cg(3)	s	tg(2)	o	og(2)	ogs(1)
cg	*	tg,cg	*	og	*	*
s		*	*	*	ogs	*
tg			*	*	ogs	*
o				*	*	*
og					*	*
ogs						*

Methods returning an ObjectGraphSlice

- ClassGraph.slice(Object, Strategy)
- ObjectGraph.slice(Strategy)
- TraversalGraph.slice(Object)
- ObjectGraphSlice(ObjectGraph,Strategy)
- ObjectGraphSlice(ObjectGraph,TraversalGraph)

Blue: constructors

Traverse method arguments

- ClassGraph
  - Object, Strategy, Visitor
- TraversalGraph
  - Object, Visitor
- ObjectGraph
  - Strategy, Visitor
- ObjectGraphSlice
  - Visitor

Traverse method arguments. Where is collection framework used?

- ClassGraph
  - Object, Strategy, Visitor / asList(Object, Strategy)
- TraversalGraph
  - Object, Visitor / asList(Object)
- ObjectGraph
  - Strategy, Visitor / asList(Strategy)
- ObjectGraphSlice
  - Visitor / asList()

Where is collection framework used?

- ObjectGraphSlice.asList()
  - a fixed-size List backed by the object graph slice.

DJ unary construction operations

- Class graph from TraversalGraph
- Class graph from all classes in package

Guidelines

```
IF you use the combination of the following pairs and triples
   for multiple traversals, fetch or gather, introduce the
   following computation saving objects:
(cg,sg,o)->ogs
(cg,sg)->tg
(cg,o)->og
(tg,o)->ogs
```

Abreviations

cg	class graph
s	strategy
tg	traversal graph
o	object
og	object graph
ogs	object graph slice

ClassGraph construction

- make a class graph from all classes in default package
  - ClassGraph()
    - include all fields and non-void no-argument methods.
  - ClassGraph(boolean f, boolean m)
    - If f is true, include all fields; if m is true, include all non-void no-argument methods.

Dynamic features of DJ ClassGraph construction

- When a class is defined dynamically from a byte array (e.g., from network) ClassGraph.addClass(Class cl) has to be called explicitly. Class cl is returned by class loader.
- ClassGraph() constructor examines class file names in default package and uses them to create class graph.

Dynamic features of DJ ClassGraph construction

- ClassGraph.addPackage(String p)
  - adds the classes of package p to the class graph. The package is search for in the CLASSPATH.
- Java has no reflection for packages. Motivates above solution.

Adding Nodes and Edges to ClassGraph

- addClass(Class cl)
  - add cl and all its members to the class graph, if it hasn't already been added.
- addClass(Class cl, boolean aF, boolean aM)
  - add cl to the class graph. If aF, add all its non-static fields as construction edges. If aM, add all its non-static non-void methods with no arguments as derived construction edges.

Combining DJ and DemeterJ

- DJ is a 100% Java solution for adaptive programming.
- DemeterJ has
  - XML style data binding facilities: code generation from schema (class dictionary).
  - Its own adaptive programming language.
- We attempt an optimal integration giving us the strong advantages of both and only few small disadvantages.

Optimal DJ and DemeterJ Integration

- Take all of DJ
- Take all of DemeterJ class dictionary notation
- Take a very tiny bit of DemeterJ adaptive programming language (basically only part that allows us to weave methods).

Combining DJ and DemeterJ

- Pros (advantages)
  - Java class generation from class dictionary (getters, setters, constructors).
  - Parser generation.
  - Better packaging of Java code into different files.
  - MUCH MORE POWERFUL.
- Cons (disadvantages)
  - No longer pure Java solution.
  - need to learn Demeter notation for class dictionaries (similar to XML DTD notation).
  - need to learn how to call DemeterJ and how to use project files.

Combining DJ and DemeterJ

- What do we have to learn about DemeterJ?
  - Class dictionaries: `*.cd` files
  - Behavior files: `*.beh` files. Very SIMPLE!
    - `A { {{ … }} }` defines methods … of class A
  - Project files: `*.prj`
    - list behavior files `*.beh` that you are using
  - Commands:
    - `demjava new, demjava test, demjava clean`

Combining DJ and DemeterJ

- What you might forget in class dictionaries that are used with DJ:
  - `import edu.neu.ccs.demeter.dj.*;`
  - visitors need to inherit from `Visitor`
  - parts of visitors need to be defined in class dictionary

Combining DJ and DemeterJ

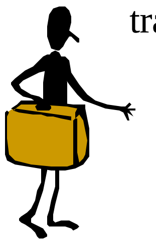
- Structuring your files
  - put reusable visitors into separate behavior files.
  - put each new behavior into a separate behavior file. Visitors that are not planned for reuse should also go into same behavior file. Update the class dictionary with required structural information for behavior to work.
  - List all `*.beh` files in `.prj` file.

Context switch

- Back to pattern language

Selective Visitor

- Intent
  - Loosely couple behavior modification to behavior and structure.
  - Would like to write an editing script to modify traversal code instead of modifying the traversal code manually.



Selective Visitor

- Could also be called:
  - Structure-shy Behavior Modification
  - Event-based Coupling

Selective Visitor

- Motivation:
  - Avoid tangling of code for one behavior with code for other behaviors.
  - Localize code belonging to one behavior.
  - Compose behaviors.
  - Modify the behavior of a traversal call (traversals only traverse).

Selective Visitor

- Applicability:
  - Need to add behavior to a traversal.

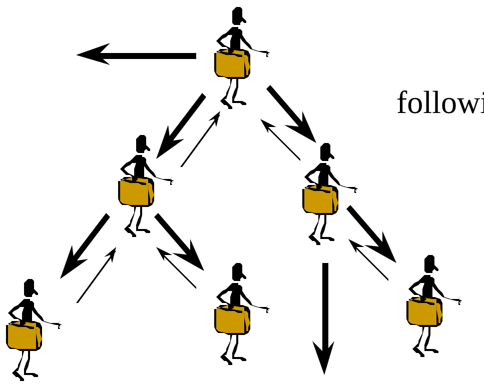
Selective Visitor

- Solution:
  – Use visitor classes and objects.
  – Pass visitor objects as arguments to traversals.
  – Either use naming conventions for visitor methods (e.g., before_A()) or extend object-oriented language (e.g. **before** A, **before** is a new key word).

Selective Visitor

- Solution:
  – before, after methods for nodes and edges in the class graph
  – Activated during traversal as follows:
    - Execute before methods
    - Traverse
    - Execute after methods

Visitor visits objects



following strategy

Visitor collects information in suitcase (variables)

Selective Visitor

- Solution: Focus on what is important.

```
SummingVisitor {
 (@ int total; @)
 init (@ total = 0; @)
 before Salary (@ total = total + host.get_v(); @)
 return (@ total @)
}
```

host is object visited

Code between (@ and @) is Java code

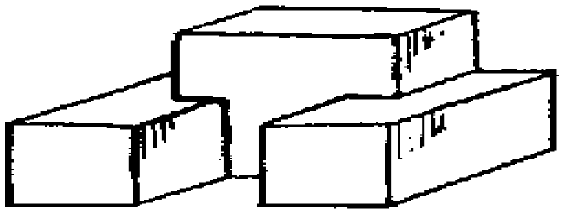
Selective Visitor

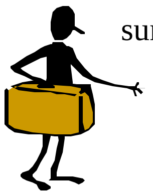
- Solution: Use of visitor

```
Company {
 traversal allSalaries(UniversalVisitor) {do S;}
 int sumSalaries()
 (@
   SummingVisitor s = new SummingVisitor();
   this.allSalaries(s);
   return s.get_return_val();
 @)
}
```

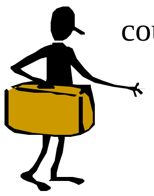
Selective Visitor

- Consequences
  - Easy behavior adjustments: Add visitor
  - Reuse of visitors





Selective Visitor



- Consequences: Easy behavior enhancement

```
Company { // enhancements in red
 traversal allSalaries(UniversalVisitor, UniversalVisitor)
   {do S;}
 (@ float averageSalaries() {
   SummingVisitor s = new SummingVisitor();
   CountingVisitor c = new CountingVisitor();
   this.allSalaries(s, c);
   return s.get_return_val() / c.get_return_val();
 }@)
}
```

Writing Programs with Strategies

Example of Adaptive Program

strategy: `from` BusRoute `through` BusStop `to` Person

```
BusRoute {
  traversal waitingPersons(PersonVisitor) {
   through BusStop to Person; } // from is implicit
  int printWaitingPersons() // traversal/visitor weaving instr.
    = waitingPersons(PrintPersonVisitor);
PrintPersonVisitor {
  before Person (@ … @) … }
PersonVisitor {init (@ r = 0 @) … }
```

Extension of Java: keywords: `traversal init through bypassing to before after` etc.

Selective Visitor

- Consequences:
  - Can reuse SummingVisitor and CountingVisitor in other applications.

Selective Visitor

- Implementation
  - Translate to object-oriented language.
  - See Demeter/Java, for example.

Selective Visitor

- Known uses
  - Propagation patterns use inlined visitor objects (see AP book).
  - Demeter/Java.
  - The Visitor Design Pattern from the design pattern book uses a primitive form of Selective Visitor.

Differences to Visitor pattern

- Focus selectively on important classes. Don’t need a method for each traversed class.
- Finer control: not only one accept method but before and after methods for both nodes and edges.

Structure-shy Object

- Intent
  - Make object descriptions for tree objects robust to changes of class structure.
  - Make object descriptions for tree objects independent of class names.

Structure-shy Object

- Could also be called:
  - Object Parsing
  - Grammar
  - Abstract=Concrete Syntax

Structure-shy Object

- Motivation
  - Data maintenance a major problem when class structure changes
  - Tedious updating of constructor calls
  - The creational patterns in the design pattern book also recognize need
  - Concrete syntax is more abstract than abstract syntax!

Structure-shy Object

- Applicability
  - Useful in object-oriented designs of any kind.
  - Especially useful for reading and printing objects in user-friendly notations. Ideal if you control notation.
  - If you see many constructor calls: think of Structure-shy Object.

Structure-shy Object

- Solution
  - Extend the class structure definitions to define the syntax of objects.
  - Each class will define a parse function for reading objects and a print visitor for printing all or parts of an object.

Structure-shy Object

- Solution
  - Start with familiar grammar formalism and change it to make it also a class definition formalism. In the Demeter group we use Wirth's EBNF formalism.
  - Use a parser generator (like YACC or JavaCC) or a generic parser.

Structure-shy Object

Parsers weave sentences into objects

Problem in OO programs: Constructor calls for compound objects are brittle with respect to structure changes.

Solution: Replace constructor calls by calls to a parser. Annotate class diagram to make it a grammar.

Benefit: reduce size of code to define objects, object descriptions are more robust

Correspondence: Sentence defines a family of objects. Adaptive program defines family of object-oriented programs. In both cases, family member is selected by (annotated) class diagram.

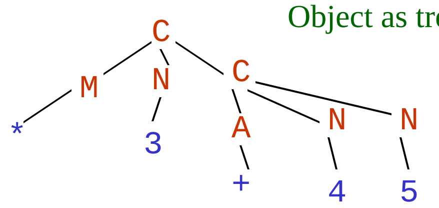
Structure-shy Object

Run-time weaving: Description

Sentence
* 3 + 4 5

Grammar
Compound ...
Simple ...
Number ...
Multiply ...
Add ...
etc.

Object as tree



Object in linear form (Constructor calls)

C M * N 3 C A + N 4 N 5

SENTENCE IS MORE ROBUST THAN OBJECT

Grammar defined by annotating UML class diagram

Structure-shy Object

- Consequences
  - more robust and shorter object descriptions
  - Need to deal with *unique readability with respect to an efficient parsing algorithm*
  - Can guarantee unique readability by adding more syntax
  - debug class structures by reading objects

Structure-shy Object

- Related patterns
  - Creational patterns in design pattern book.
  - Interpreter pattern uses similar idea but fails to propose it for general object-oriented design.
  - Structure-shy Object useful in conjunction with Prototype pattern.

Structure-shy Object

- Known uses
  - Demeter Tools since 1986, T-gen, applications of YACC, programming language Beta and many more.

Structure-shy Object

- References
  - Chapters 11 and 16 of AP book describe details.
- Exercise
  - Use your favorite grammar notation and modify it to also make it a class graph notation.