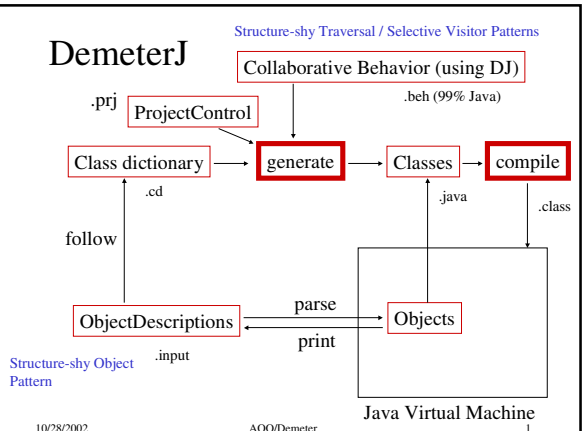


Growth plan pattern

- Intent
  - Build your adaptive programs incrementally. Use structural and behavioral simplifications which allow, ideally, for growth by addition and refinement.
- Could also be called:
  - Evolutionary development

Earlier Patterns

Four Patterns

- Structure-shy Traversal
- Selective Visitor
- Structure-shy Object
- Class Graph

How your project/hw directory should look like

- `/personality-proj`
  - `/growth-phase1`
  - `/growth-phase2`
  - `/growth-phase3`
  - …
- Each directory contains a running program.

Growth plan

- Motivation: It is useful to have at all times a simplified version of the program running.
  - good for your self-confidence
  - good for your customers: feedback

  Build applications in growth phases, where a phase *next* can do more than a previous phase *previous*.

Growth plan

- Motivation (continued): We want to build *next* as much as possible out of *previous* by adding or refining, not by modifying *previous*. *next* ideally reuses the test inputs of *previous.*
- Application: Use this pattern when you build applications involving more than a small number of classes (say 5).

Growth plan

- Solution: A good strategy is to build a relative big chunk of the class dictionary of the application and to test it with several input sentences. Then build a structural shrinking plan (whose inverse is a structural growth plan) consisting of a decreasing (in size) sequence of class dictionaries.

Growth plan

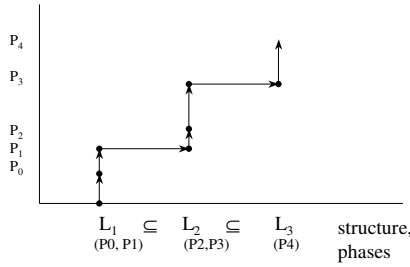
- Solution (continued): For the smallest class dictionary you should be able to implement some interesting behavior. For each phase in the structural growth plan you implement increasingly more complex behavior.

Growth plan

- Solution (continued): The structural growth steps should ideally be language extending so that the inputs of phase i also are legal inputs for phase i+1.
- The behavioral growth steps should ideally be additive and should require only small modifications.

Growth plan

behavior phases



OS simulation example

- Phase 1:
  - Structure: Start with full class dictionary and use the following slices: tg1 = from FileSystem to * and tg2 = from Commands to CreateEmptyFile.
  - Behavior: Main.cdir. Commands::process(tg2){ CommandVisitor cv = ...; tg2.traverse(this,cv);} CommandVisitor::before(CreateEmptyFile h){ create SimpleFile sf; addElement(sf);}

OS simulation example

- Phase 1:
  - Why useful? What does it test? We can check whether simple files that are created anywhere in the input, are properly added to the root directory.
  - Program already shows some of the right behavior: if we only have touch commands.
  - But also on other inputs it shows useful behavior.

Phase 1: class dictionary

```
FileSystem = <root> CompoundFile EOF.
File : SimpleFile | CompoundFile common <f>
   FileName.
SimpleFile = "simple".
CompoundFile = "compound" <contents> PList(File)
   [<parent> CompoundFile].
```


Phase 1: class dictionary

```
Commands = List(Command) EOF.        Command : Simple.
Simple : MakeDirectory | ChangeDirectoryUp |
   ChangeDirectoryDown |
RecursiveCopy | DiskUsage | Find | Echo |
SymbolicLink | RemoveDirectory | CreateEmptyFile |
   RemoveFile.
MakeDirectory = "mkdir" DirectoryName.
ChangeDirectoryUp = "cd ..".
ChangeDirectoryDown = "cd" DirectoryName.
RecursiveCopy = "cp -r ../* ." .
DiskUsage = "du .".
SymbolicLink = "ln -s" <from> FileName <to> FileName.
RemoveDirectory = "rmdir" DirectoryName.
```


Phase 1: class dictionary

```
// "touch f" creates an empty file called f.
CreateEmptyFile = "touch" FileName.
RemoveFile = "rm" FileName.
Find = "find . -name" DirectoryName "-print".
Echo = "echo" Message.
FileName = Ident.
DirectoryName = Ident.
Message = String.
PList(S) ~ "(" {S} ")".
List(S) ~ {S}.
Main = .
```


Desired behavior (example)

```
FileSystem fs =
   new FileSystem(
     new CompoundFile(new FileName(
    new Ident ("root")),
  new File_PList()
       )
   );
File_PList filelist1 = fs.get_root().get_contents();
CompoundFile a =
     new CompoundFile( new FileName( new Ident ("a")),
       new File_PList());
filelist1.addElement(a);
```

Desired behavior (example)

```
CompoundFile b =
     new CompoundFile(
       new FileName(
         new Ident ("b")
     ),
     new File_PList());
filelist1.addElement(b);
```


Desired behavior (example)

```
   File_PList filelist2 = a.get_contents();
       CompoundFile c =
         new CompoundFile(
           new FileName(
             new Ident ("c")
         ), new File_PList());
   filelist2.addElement(c);
```


Phase 1

```
Commands {
{{
  void process(TraversalGraph where) {
     CommandVisitor cV =
        new CommandVisitor();
     where.traverse(this, cV);
  }
}}
}
```


Phase 1

```
CommandVisitor { {{
   void before(CreateEmptyFile host){
      SimpleFile sf = SimpleFile.parse("simple " +
       (Ident) Main.cg.fetch(host,
              "from CreateEmptyFile to" + Main.FIdent);
      // host.get_filename().get_ident());
      // SimpleFile sf = new SimpleFile(
      //   host.get_filename());
     Main.cdir.get_contents().addElement(sf);
     System.out.println(" CreateEmptyFile ");
   }
}} }
```

Phase 1

```
Main {
(@
  static CompoundFile cdir;
  static ClassGraph cg;
  static String FIdent = " edu.neu.ccs.demeter.Ident ";
  static public void main(String args[]) throws
  Exception {
    Commands cs = Commands.parse(System.in);
    cs.print();
    System.out.println();
    FileSystem fs = FileSystem.parse(
    "compound () root");
```


Phase 1: Main continued

```
cdir = fs.get_root();
cg = new ClassGraph(true, false);
ClassGraph cgCommandsWithoutTail =
  new ClassGraph(Main.cg,
     "from Commands bypassing -> *,tail,* to *");
cs.process(new TraversalGraph
   ("from Commands to *",
      cgCommandsWithoutTail));
```


Input for testing phase 1

```
touch a
touch b
mkdir x
touch c
cd x
touch d
```


OS simulation example

- Phase 2:
  - Structure: Use larger part of class dictionary: tg2 = from Commands to {CreateEmptyFile, MakeDirectory}.
  - CommandVisitor::before(MakeDirectory h){ create CompoundFile cf; addElement(cf);}

OS simulation example

- Phase 2:
  - Why useful? What does it test? We can check whether simple files and directories that are created anywhere in the input, are properly added to the root directory.
  - Program already shows some of the right behavior: if we only have touch and mkdir commands.
  - But also on other inputs it shows useful behavior.

Input for testing phases 1 and 2

```
touch a
touch b
mkdir x
touch c
cd x
touch d
```


Phase 2

```
CommandVisitor {
{{
 void before(MakeDirectory host){
     Ident id =
       (Ident) Main.cg.fetch(host,
          "from MakeDirectory to" + Main.FIdent);
     CompoundFile cf = CompoundFile.parse("compound ()" + id);
     cf.set_parent(Main.cdir);
//       new CompoundFile(
//       new FileName(host.get_directoryname().get_ident()),
//       new File_PList(),Main.cdir);
     Main.cdir.get_contents().addElement(cf);
     System.out.println(" MakeDirectory ");
   }
```


What is next?

- ChangeDirectoryDown
- ChangeDirectoryUp
- Which one is easier to test? It is very important that we can test each phase to get immediate gratification whether we did it right.

OS simulation example

- Phase 3:
  - Structure: Use larger part of class dictionary: tg2 = from Commands to {CreateEmptyFile, MakeDirectory, ChangeDirectoryDown}.
  - Change CommandVisitor: entry for ChangeDirectoryDown; Search through current directory to find directory to enter. Update Main.cdir.

Input for testing phases 1, 2, 3

```
touch a
touch b
mkdir x
touch c
cd x
touch d
```


OS simulation example

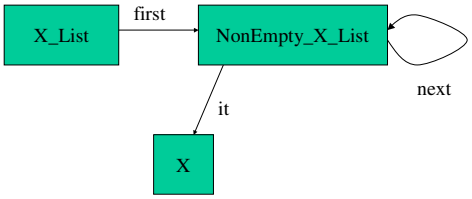
- Phase 4:
  - Structure: Use larger part of class dictionary: tg2 = from Commands to {CreateEmptyFile, MakeDirectory, ChangeDirectoryDown, ChangeDirectoryUp}.
  - Change class dictionary: parent field.
  - Change display: infinite loop: 2 options.
  - Change CommandVisitor: entry for ChangeDirectoryUp; update entry for MakeDirectory: maintain parent.

Input for testing phase 4

```
mkdir x
cd x
mkdir y
cd y
touch d
cd ..
touch c
```

For phases 3 and 4: List Structure

- List(X)



Iterating through a DemeterJ list

- Have an X_List xlist;

```
java.util.Enumeration en = xlist.elements();
while (en.hasMoreElements()) {
  if (e.equals((X) en.nextElement()))
     { found = true; }
  found = false;
}
```

Enumeration Interface
boolean hasMoreElements();
Object nextElement();

Iterating through a DemeterJ list: in class X_List

```
public java.util.Enumeration elements() {
  return new X_List(first);
}
public Object nextElement() {
  X car = first.get_it();
  first = first.get_next();  return (Object) car;
}
```


Iterating through a DemeterJ list: in class X_List

```
public boolean hasMoreElements()
   { return (first != null); }
```

Enumeration interface is old fashioned

- Use Iterator interface instead
  - boolean hasNext();
  - Object next();
  - void remove(); (optional operation)
- Compare to

Enumeration Interface
boolean hasMoreElements();
Object nextElement();

Enumeration interface is old fashioned

- ListIterator is a subinterface of Iterator
  - boolean hasNext();
  - Object next();
  - void remove(); (optional operation)
  - add, hasPrevious, previousIndex, nextIndex, previous, set

Java documentation

- The functionality of the Enumeration interface is duplicated by the Iterator interface. … shorter method names ... New implementations should consider using Iterator in preference to Enumeration.

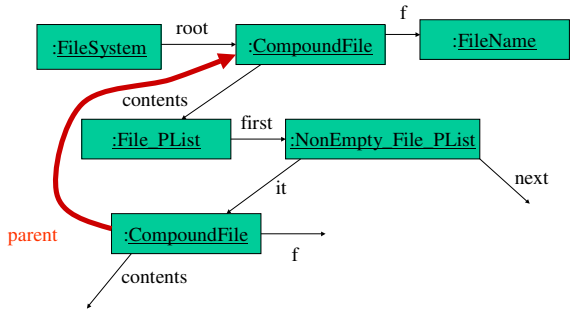
Traversal with a ListIterator

```
void traverse_maxSize(IntegerRef m){
    for (ListIterator
  i=this.listIterator(); i.hasNext();)
  {
      DirectoryEntry de =
        (DirectoryEntry) i.next();
      de.traverse_maxSize(m);
    }
```

How can you get an Iterator?

- Interface Collection:
  - Iterator iterator();
- Example:
  - class Vector implements interface List
  - interface List extends interface Collection
  - Therefore: Use the Iterator interface to go through a vector

Change `display()` method



Testing for Type in Java

```
if (f instanceof CompoundFile) {
  cf = (CompoundFile) f;
  ...
 }
```

USE SPARINGLY!

OS simulation example

- Phase 5:
  - Structure: Use larger part of class dictionary: tg2 = from Commands to {CreateEmptyFile, MakeDirectory, ChangeDirectoryDown, ChangeDirectoryUp, DiskUsage}.
  - Change CommandVisitor: entry for DiskUsage: requires itself a traversal.

Output to expect

- `In a DemeterJ diretory: du .`

```
./gen/classes
./gen
.
```


OS simulation example

- Phase 5:
  - Traversal in visitor: from CompoundFile to File
  - Visitor: before CompoundFile: print path from current directory

Growth plan

- Solution (continued): For the smallest class dictionary you should be able to implement some interesting behavior. For each phase in the structural growth plan you implement increasingly more complex behavior. The structural growth steps should fall into one or both of the following categories:

cd = class dictionary

Growth plan

- Solution (continued):
  - weakly object extending cd transformations
    - The next class dictionary defines more objects but does not invalidate any existing objects. What runs now should run later. Reuse of test objects.
  - language extending cd transformations
    - The next cd defines a super language of the language of the current cd.

Object-extending transformations

- relations on class graphs, associated with transformations, fundamental for reuse
  - object-equivalence
    - preserves the set of objects
  - weak extension
    - enlarges the set of objects
  - extension
    - enlarges and augments the set of objects

Part clusters

- What can be put into parts?
- PartClusters of a class *v* is a list of pairs, one for each induced part of *v*. Each pair consists of the part name and the set of construction classes whose instances can be assigned to the part
- $PartClusters_{Furnace}$(TempSensor) = {temp {Kelvin, Celsius}, trigger {Integer}}

Object-equivalence

- Let G_1 and G_2 be two class graphs. G_1 is object-equivalent to G_2 if for the concrete classes VC_1 of G_1 and the concrete classes VC_2 of G_2:
  - $VC_1 = VC_2$
  - and for all *v* in VC_1:

    $PartClusters_{G1}(v) = PartClusters_{G2}(v)$.

Covered

- Let PC_1 and PC_2 be two part clusters. PC_1 is covered by PC_2 if for each pair (l,T_1) in PC_1 there exists a pair (l,T_2) in PC_2 such that $T_1 \subseteq T_2$.
- Tightly covered means: covered and $|PC_1| = |PC_2|$.

Weak extension

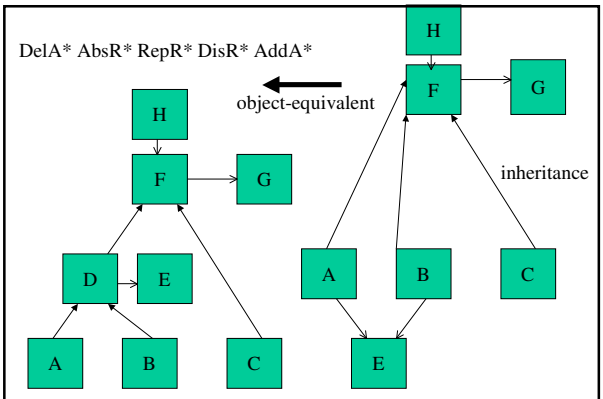
- Let G_1 and G_2 be two class graphs. G_1 is a weak extension of G_2 if for the concrete classes VC_1 of G_1 and the concrete classes VC_2 of G_2:
  - $VC_1 \subseteq VC_2$ and for all v in VC_1:
  - $PartClusters_{G1}(v)$ is tightly covered by $PartClusters_{G2}(v)$.

Extension

- Let G_1 and G_2 be two class graphs. G_1 is an extension of G_2 if for the concrete classes VC_1 of G_1 and the concrete classes VC_2 of G_2:
  - $VC_1 \subseteq VC_2$ and
  - for all v in VC_1: $PartClusters_{G1}(v)$ is covered by $PartClusters_{G2}(v)$.

Properties

- The three class graph relations have the following inclusion properties:
  - object-equivalence $\subseteq$
  - weak-extension $\subseteq$
  - extension



Primitive Transformations

- Addition of Abstract Class (AddA)
- Deletion of Abstract Class (DelA)
- Abstraction of Common Reference (AbsR)
- Distribution of Common Reference (DisR)
- Replacement of Reference (RepR)

object-equivalence = DelA* AbsR* RepR* DisR* AddA*.

Primitive Transformations

- Addition of Abstract Class (AddA)
  - adds an abstract class *u* and subclass edges outgoing from *u*. *u* must not have any outgoing construction edges.
- Deletion of Abstract Class (DelA)
  - inverse of AddA. Deletes an abstract class *u* and all its subclass edges. *u* must not have any incoming construction or subclass edges nor any outgoing construction edges.

Primitive Transformations

- Abstraction of Common Reference (AbsR)
  - moves a construction edge common to a set of sibling classes up to their direct superclass.
- Distribution of Common Reference (DisR)
  - moves a construction edge to the direct subclasses.

Primitive Transformations

- Replacement of Reference (RepR)
  - reroutes a construction edge (v,l,u_1) to a new target (v,l,u_2) where u_1 and u_2 have the same set of concrete subclasses.

Primitive Transformations

- Addition of Concrete Class (AddC)
- Generalization of Reference (GenR)
- Addition of Reference (AddR)
- Equiv = object equivalence

weak-extension =
(Equiv((GenR)Equiv)*AddC*

extension =
(Equiv((AddR|GenR)Equiv)*AddC*

Primitive Transformations

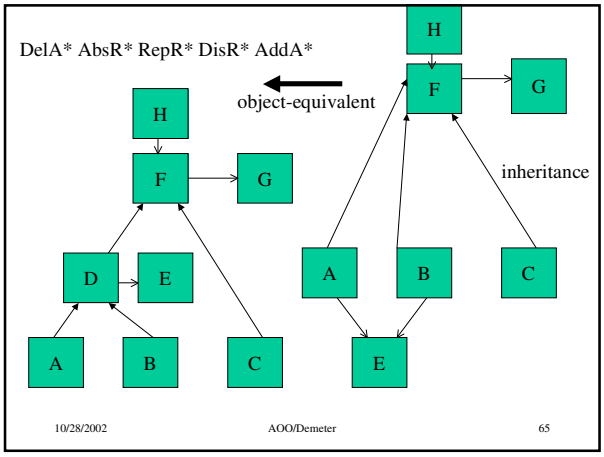
- Addition of Concrete Class (AddC)
  - adds an "empty" concrete class
- Generalization of Reference (GenR)
  - reroutes a construction edge (v,l,u_1) to a new target (v,l,u_2), where u_2 is a direct superclass of u_1.

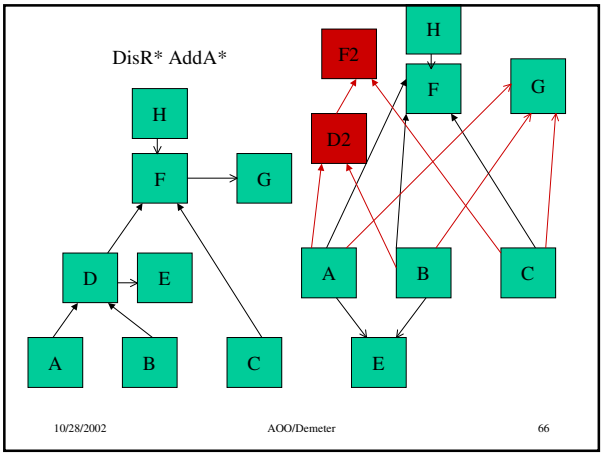
Primitive Transformations

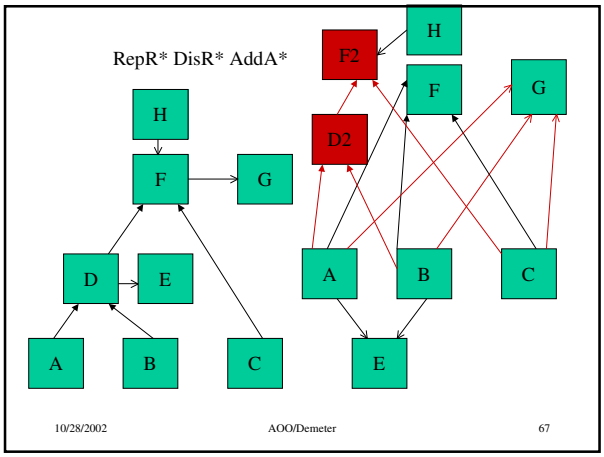
- Addition of Reference (AddR)
  - adds a new construction edge between existing vertices of the class graph.

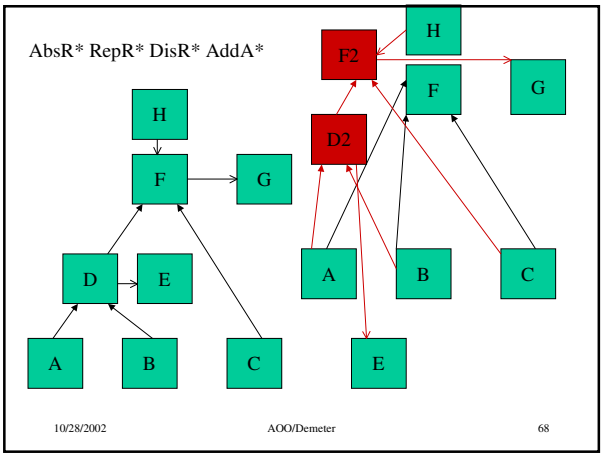
Warning

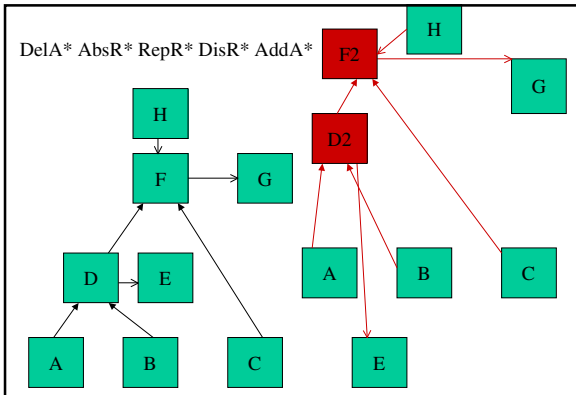
- In the following viewgraphs is-a and has-a edges should be switched.


DelA* AbsR* RepR* DisR* AddA*
object-equivalent
inheritance
H
F
G
D
E
A
B
C
10/28/2002
AOO/Demeter
65


DisR* AddA*
F2
D2
H
F
G
D
E
A
B
C
10/28/2002
AOO/Demeter
66


RepR* DisR* AddA*
F2
D2
H
F
G
D
E
A
B
C
10/28/2002
AOO/Demeter
67


AbsR* RepR* DisR* AddA*
F2
D2
H
F
G
D
E
A
B
C
10/28/2002
AOO/Demeter
68



Connections

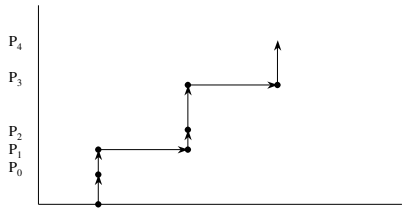
- weak extension implies language extension
  - If two cds are in a weakly object-extending relationship they can be brought to a language extending relationship with appropriate syntax.
- object-equivalent extension implies language-equivalent extension
  - similar

Growth plan

- behavior transformations should ideally extend the program by addition instead of modifying it:
  - use inheritance between visitor classes
  - add methods, traversals, visitors
  - refine methods and visitors
  - refine strategies (add more constraints)

Growth plan

behavior phases



Growth plan

- Consequences: Following Growth plan has a number of benefits:
  - Gradual building of confidence in your software development skills.
  - Show prototypes to your customers.
  - Simplified testing. Find earliest phase where a bug shows up.
  - faster compilation and generation

Growth plan

- Implementation: Create separate directories for each growth phase. Document changes.

  See chapter 13 in AP book for study of class graph extension. Also follow the pages listed under index entry *growth plan*.

Example

Same adaptive program for Terminal Buffer Rule checking works for both phases. Faster for phase 1.

phase 1

```
Cd_graph = <first> Adj.
Adj = <vertex> Vertex <ns> Construct ".".
Construct = "=" <l1> Labeled_vertex <l2> Labeled_vertex.
Labeled_vertex = "<" <label_name> Label ">"
  <class_name> Vertex.
Vertex = <name> Ident. Label = <name> Ident.
```

phase 2
NOT even an extension

```
Cd_graph = <first> Adj <rest> Adj_list.
Adj = <vertex> Vertex <ns> Neighbors ".".
Neighbors : Construct | Alternat.
Construct = "=" <c_ns> Any_vertex_list.
Labeled_vertex = "<" <label_name> Label ">"
  <class_name> Vertex.
Vertex = <name> Ident.
Any_vertex_list = <first> Any_vertex <rest> Any_vertex_list.
Any-Vertex : Labeled_vertex | Syntax_vertex.
...
```


Conclusion

- Using Adaptive Programming you can apply the Growth Plan pattern effectively. You can start with simple structures and generalize your program to more general structures easily.

Further information

- Paul Bergstein's OOPSLA 91 paper
- Walter Huersch's Ph.D. thesis
- Linda Seiter's Ph.D. thesis

End of lecture