

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 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. 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
- $PartClusters_{Furnace}(\text{TempSensor}) = \{ \text{temp} \{ \text{Kelvin}, \text{Celsius} \}, \text{trigger} \{ \text{Integer} \} \}$

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 :
$$PartClusters_{G_1}(v) = PartClusters_{G_2}(v).$$

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 $T_1 \hat{=} T_2$.
- 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 :
 - $PartClusters_{G_1}(v)$ is tightly covered by $PartClusters_{G_2}(v)$.

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 : $PartClusters_{G_1}(v)$ is covered by $PartClusters_{G_2}(v)$.

Properties

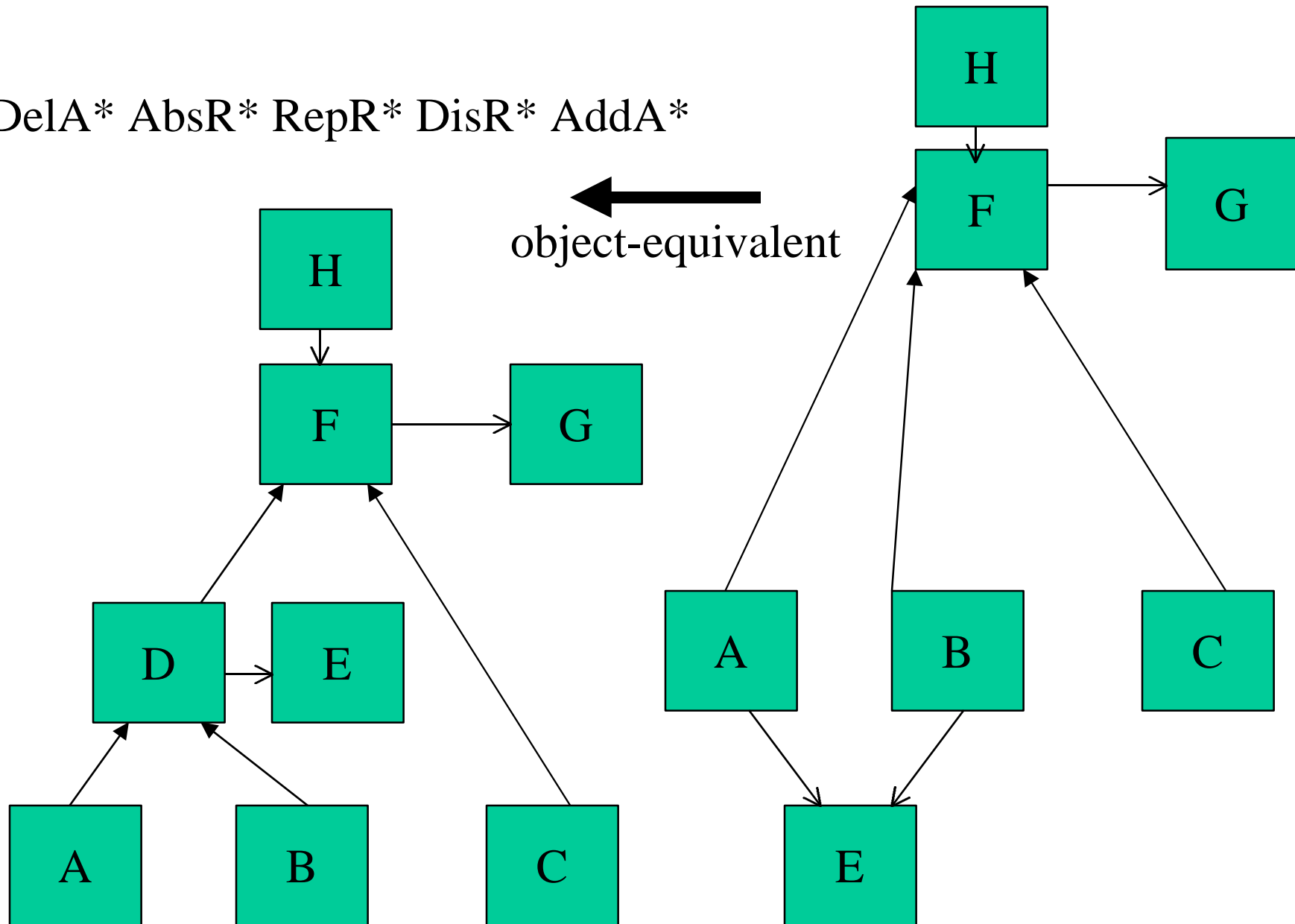
- The three class graph relations have the following inclusion properties:
 - object-equivalence \subseteq
 - weak-extension \subseteq
 - extension

Primitive Transformations

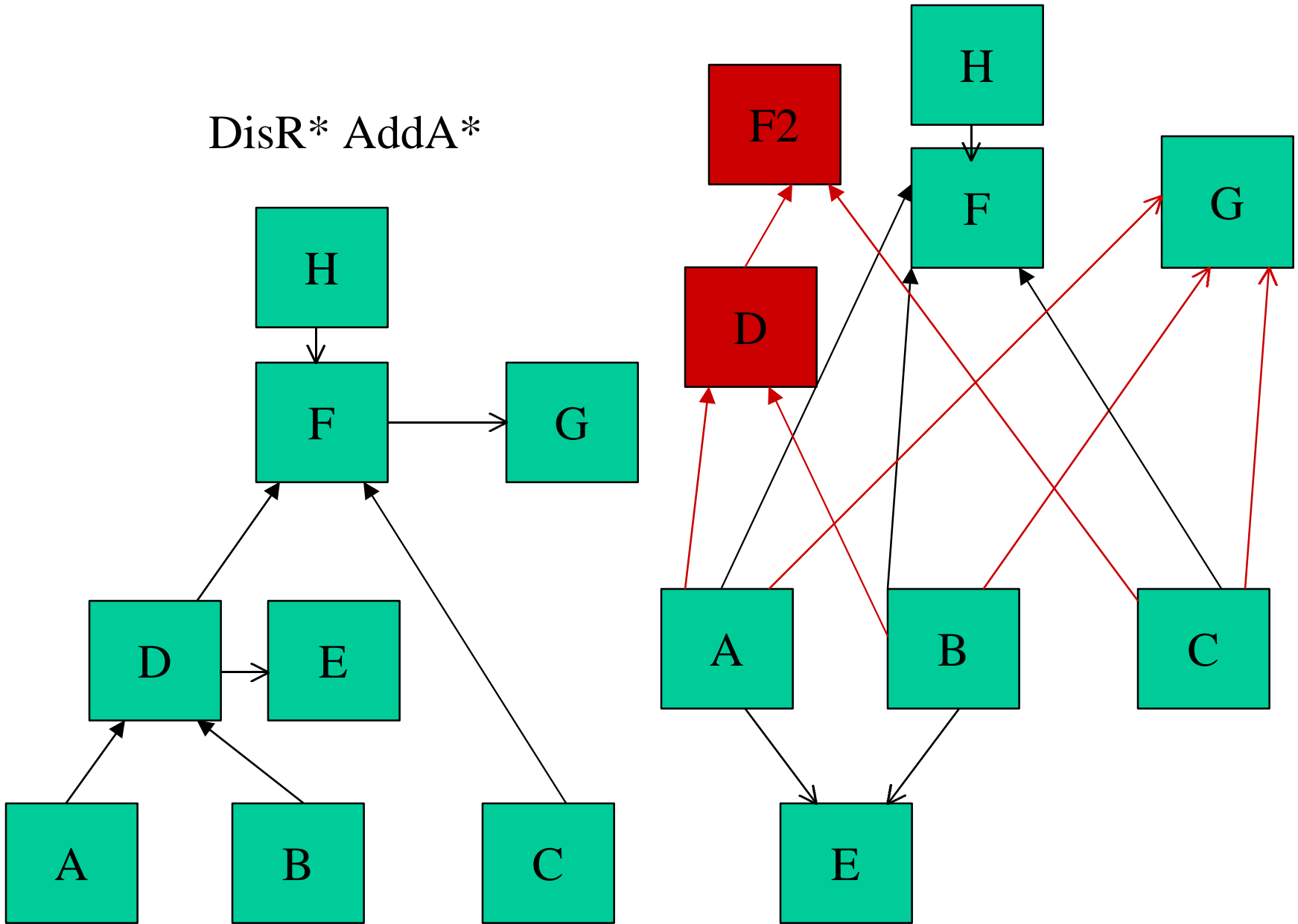
- Addition of Abstract Class (AddA)
- Deletion of Abstract Class (DelA)
- Abstraction of Common Reference (AbsR)
- Distribution of Common Reference (DisR)
- Replacement of Reference (RepR)

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

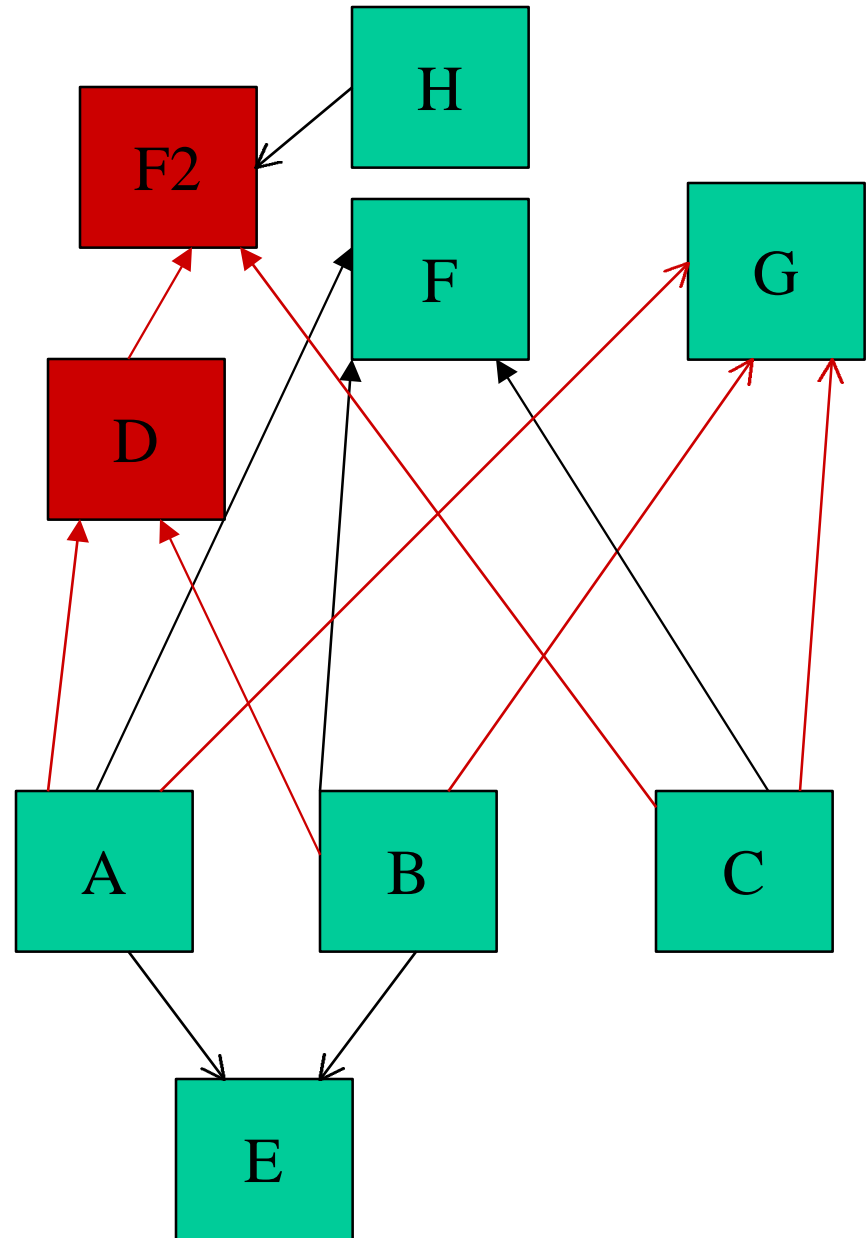
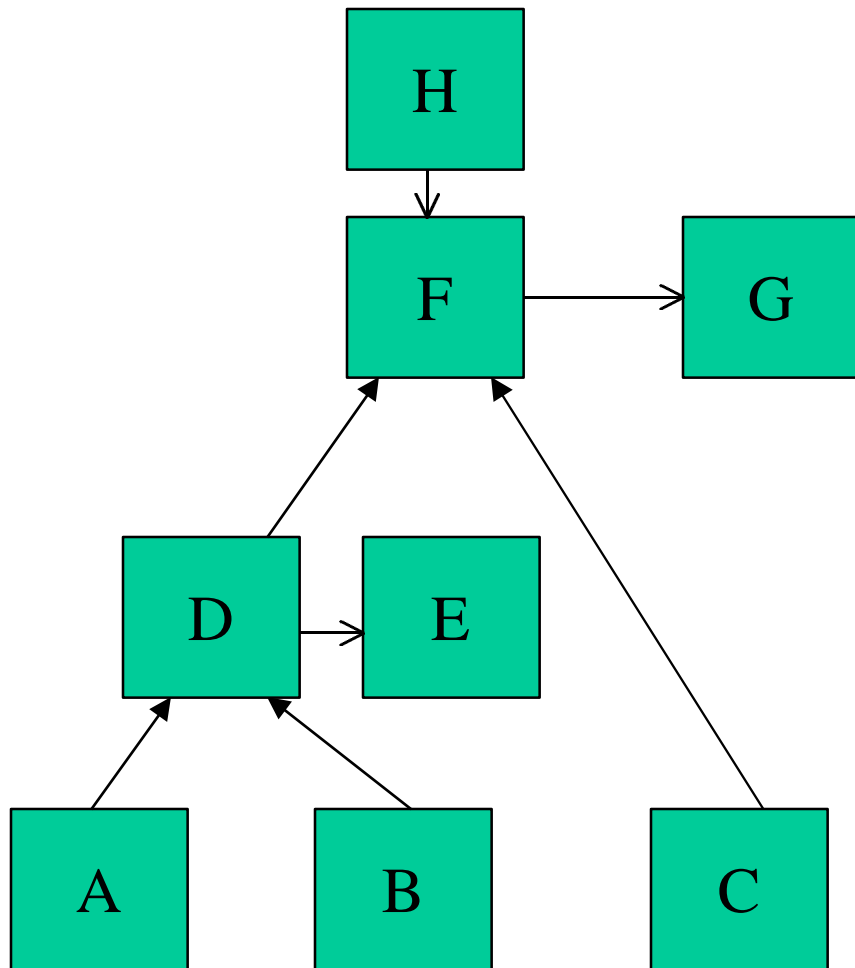
DelA* AbsR* RepR* DisR* AddA*



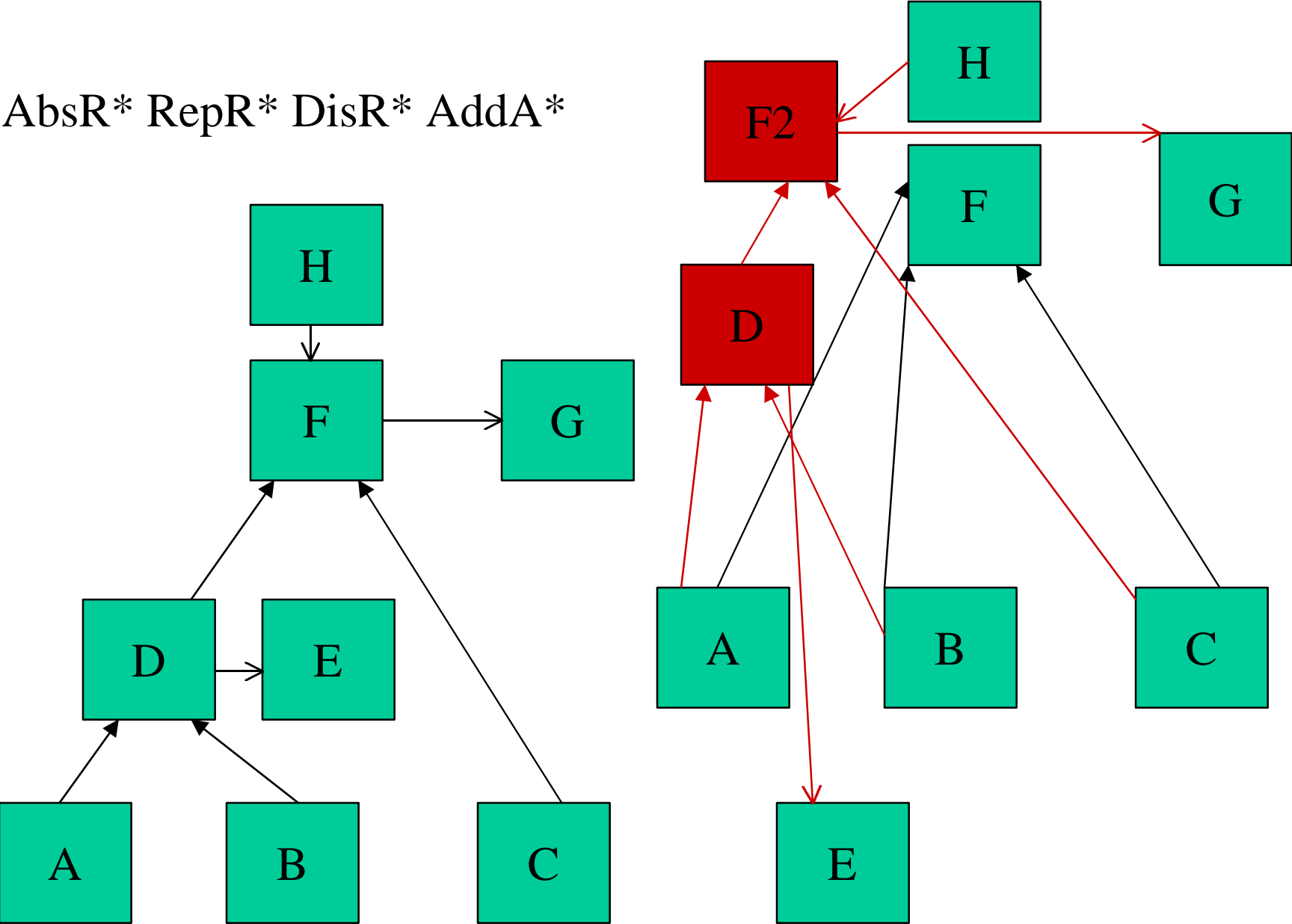
DisR* AddA*



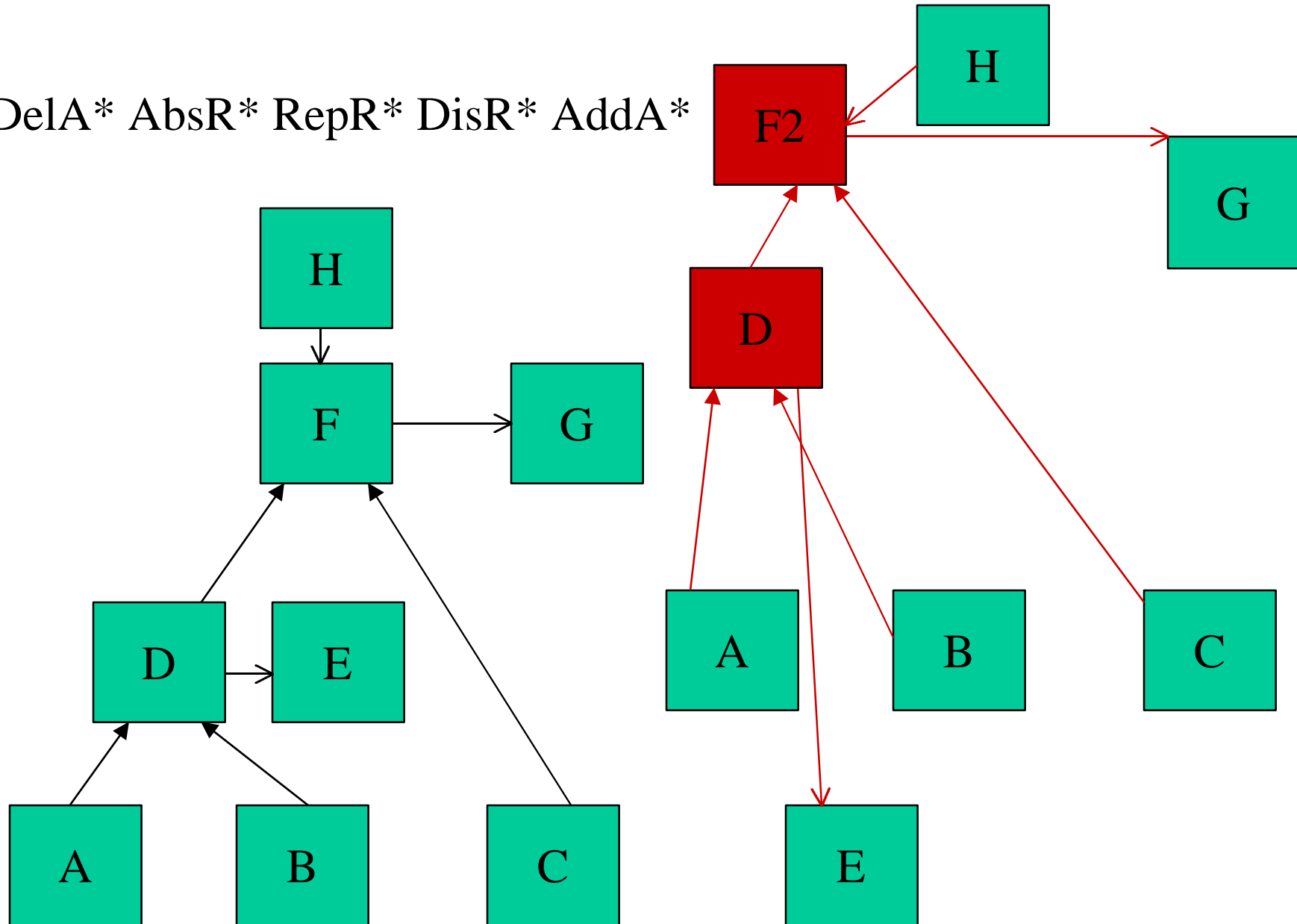
RepR* DisR* AddA*



AbsR* RepR* DisR* AddA*



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 =

$(\text{Equiv}((\text{GenR})\text{Equiv}) * \text{AddC} *)$

extension =

$(\text{Equiv}((\text{AddR}|\text{GenR})\text{Equiv}) * \text{AddC} *)$

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.

Connections

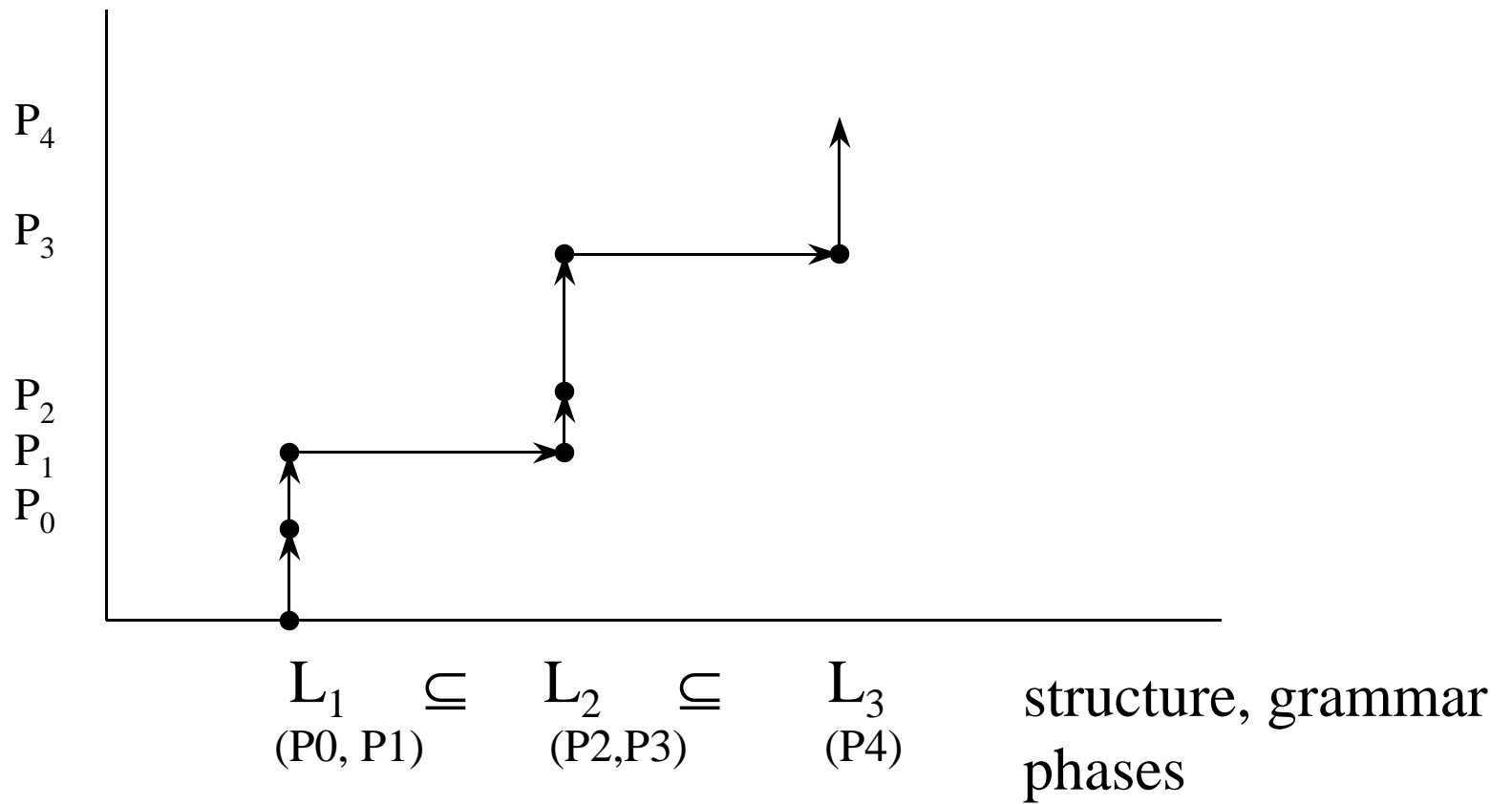
- weak extension implies language extension
 - If two cds are in a weakly object-extending relationship they can be brought to a language extending relationship with appropriate syntax.
- object-equivalent extension implies language-equivalent extension
 - similar

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.

```
Cd_graph = <first> Adj. phase 1  
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.
```

```
Cd_graph = <first> Adj phase 2 <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.  
...
```

Further information

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

End of lecture

Topic switch

- Relationship between interface class graph/strategy graph and concrete class graph
- Should be refinement relationship
- Has implication on traversal history
 - For all traversal histories H in concrete class graph there is an interface class graph object whose traversal history is an abstraction of H .

Abstraction of traversal history

- Assume embedded strategy
- Can get traversal history in concrete class graph by adding new nodes not in the interface class graph
- Motivation: computations in two graphs are sufficiently similar for being correct in both.

Refined Path Set

- Strategy graph S with source s and target t of a base graph G . $Nodes(S)$ subset $Nodes(G)$ (Embedded strategy graph).
- S defines *refined path set* in G as follows:
 $RefPathSet_{st}(G, S)$ is the set of all $s-t$ paths in G that are refined expansions of any $s-t$ path in S .

Refined expansion

- Let S be a strategy graph, let G be a base graph with $Nodes(S) \dot{\cap} Nodes(G)$. Given a strategy-graph path $p = \langle a_0 a_1 \dots a_n \rangle$, we say that a path p' in G is a *refined expansion* of p if there exist paths p_1, \dots, p_n in G such that $p' = p_1 \cdot p_2 \dots p_n$ and: For all $0 < i < n + 1$, $Source(p_i) = a_{i-1}$ and $Target(p_i) = a_i$ and the interior of p_i $0 < i < n + 1$ does not contain nodes from S .

Alternate Definition of Refinement

- G is a refinement of S iff and only if $RefPathSet_{st}(G,S) = PathSet_{st}(G,S)$