Adaptive Programming

Prof. Dr.-Ing. P. Kroha
University of Technology Chemnitz
Department of Computer Science, Germany

Outline

1. Problem
2. Motivation
3. Example of a class hierarchy and query evolution
4. Adaptive object-oriented programming
5. The Law of Demeter
6. Class dictionary, class graph
7. Traversal strategies
8. Propagation pattern
9. Visitor
10. Conclusion

Recommended Reading

Lieberherr, K.:

Adaptive Object-Oriented Software:

The Demeter Method with Propagation Patterns.

PWS Publishing Company, 1996.

Demeter Research Group:

http://www.ccs.neu.edu/research/demeter/DemeterJava

1. What is the Problem?

Object-oriented technology **has not met its expectations** when applied to real business applications
partly due to the fact that
there is **no place where to put higher-level operations which affect several objects**.

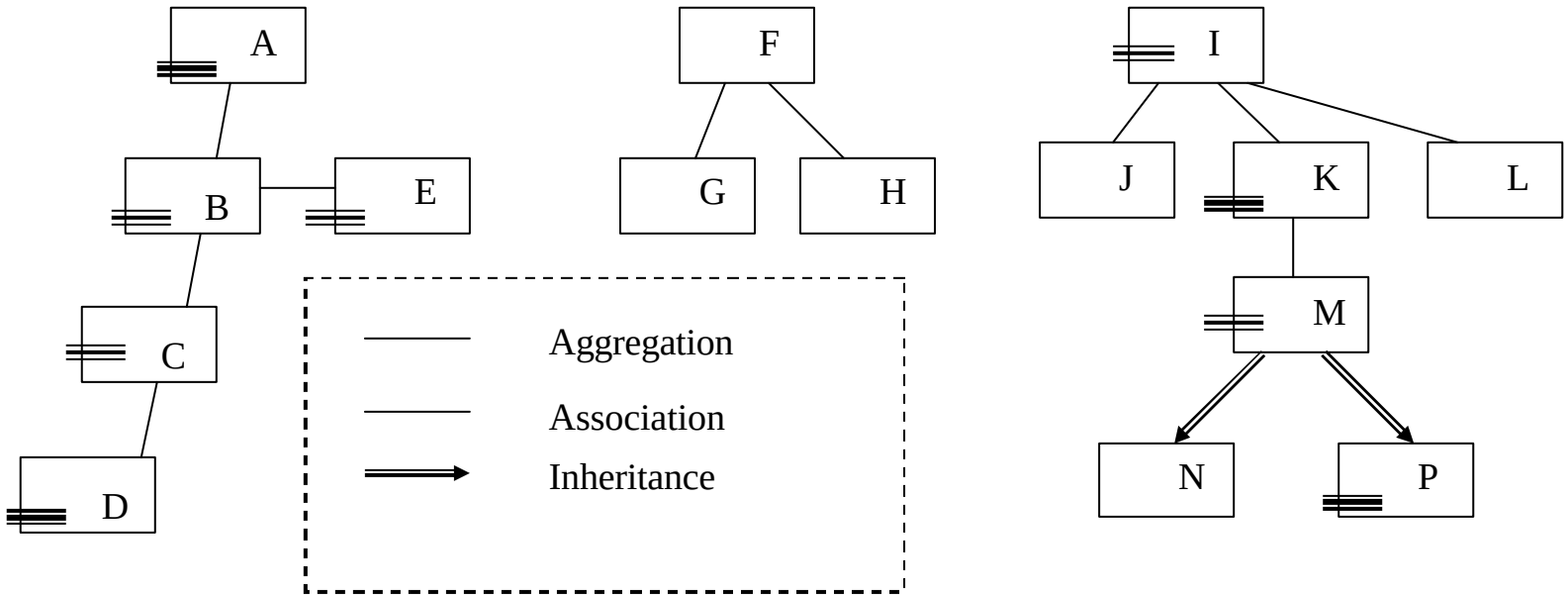
… if built into the classes involved, it is impossible to get an overview of the control flow.

[Lauesen, IEEE Software, April '98]

2. Motivation

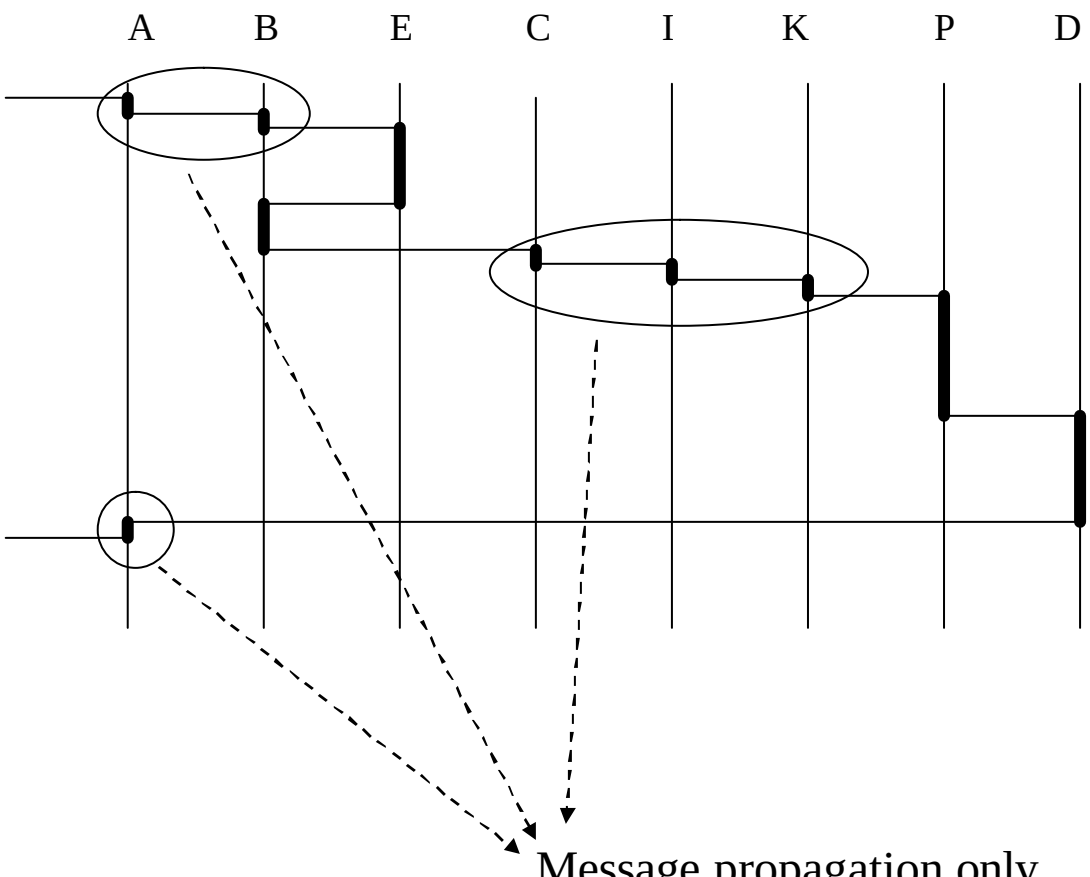
- Object-oriented programs contain too much redundant application-specific information.
- An operation (a query, a concatenation of invoked methods) is spread in methods of various classes. Each of such methods has to know all details about the class hierarchy to know the destination to what it should send a message for the query processing to be continued.
- We need to be able to specify only essential elements, in a way that can adapt accordingly to the changing class hierarchy.

A Spread Operation - Classes



The operation is spread over the class hierarchy without any systematic approach

A Spread Operation – Sequence Diagram



Because there was not a systematical approach in spreading the operation into methods, each object has to know exactly to whom it should send the message.

The message propagation cannot be generated automatically.

Any change in class hierarchy can cause a necessity to modify programs.

=> Expensive maintenance

Writing Operations in OO Languages

- When OO programming languages (C++, Java) are used for querying (for describing operations) they navigate in the schema.
- Operations in OO programming languages have to contain a hardwired navigation path.
- For the computation of an operation, the programmer has to describe:
  - the sequence of messages (event-trace) in the sense of a sequence diagram
  - a set of methods spread in classes that will accept and execute these messages when concatenated by a message flow

Maintenance of OO Programs

- The sequence of messages of an operation depends on the underlying class hierarchy.
- Should the class hierarchy be changed because of the system evolution, usually also the sequence of messages must be changed, because some navigation paths have changed.

  => Operations have always to be checked

  and usually reprogrammed

  => Expensive maintenance

Navigation in OO Programs

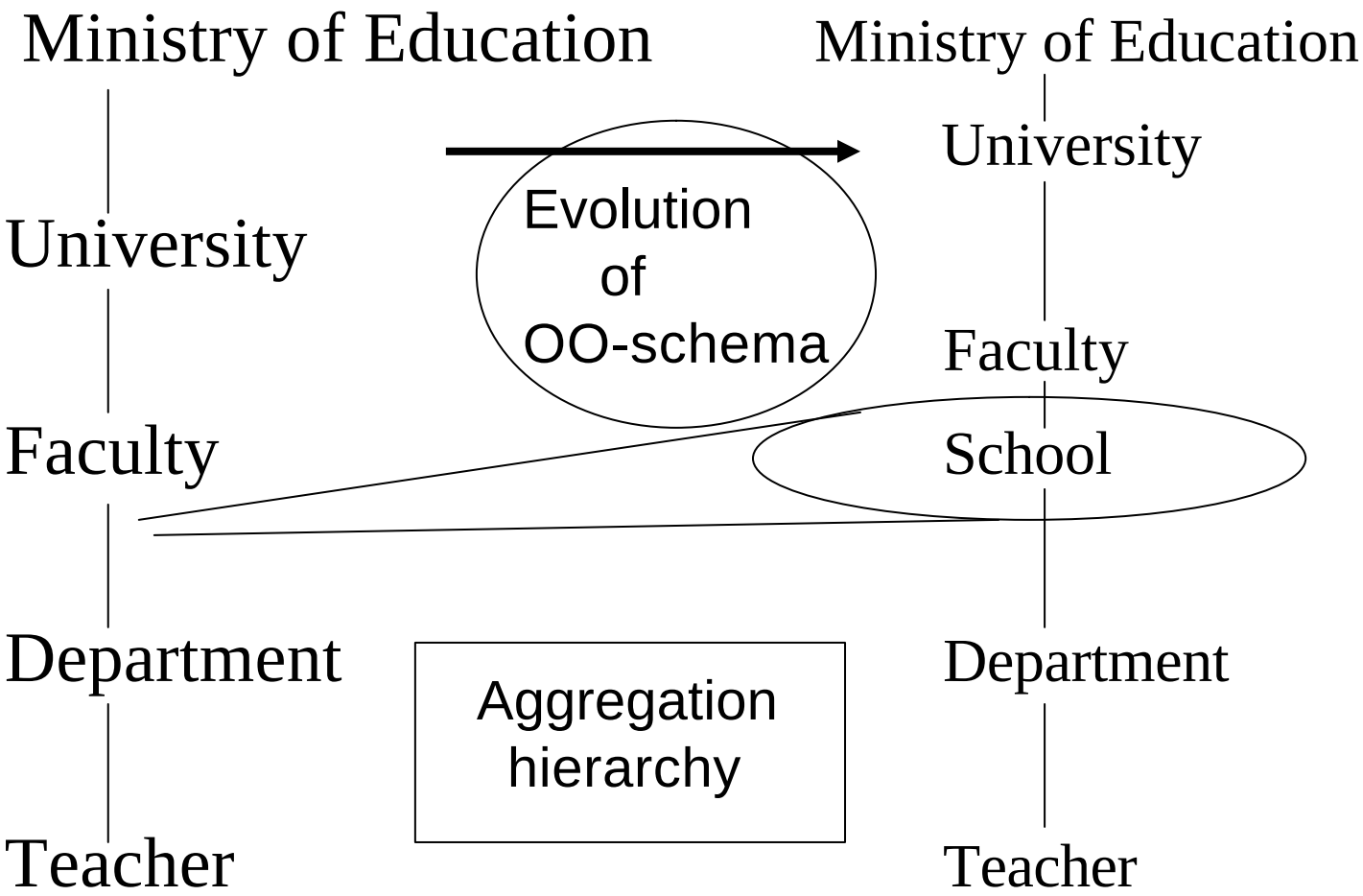
- Navigation in OO programs,

  which is contained in an operation, can be given by:
  - a simple aggregation => a simple concatenation of method invocations:

    Object . M1(Par1) . M2(Par2) . M3 . M4 . M5(Par5)
  - an aggregation using containers:

    For EachContainerElement do … (example follows)
- => any change in the class structure can violate it

3. Example of Evolution



Example of a Query

- A query of the minister:

  “How many teachers do we have at universities?”
- The query implementation in C++ depends on the class hierarchy.
- After every structural change, all queries must be reprogrammed ➔ VERY UNPLEASANT
- Structural changed in some political and organisation systems are very frequent
  - => very expensive maintenance
  - => decreasing reliability

Example - Query before Evolution

```
MinistryOfEducation ← Minister (How many teachers do we have?)
    TotalTeachers : = 0
    P :- MinistryOfEducation . ListOfUniversties
or  Each P.University do University← MinistryOfEducation (How many teachers do you have
  Q :- University.ListOfFaculties
or Each Q.Faculty do Faculty ← University (How many teachers do you have?)
  R :- Facultyl.ListOfDepartments
or Each R.Department do Department ← School (How many teachers do you have?)
          S:-Department.ListOfTeachers
or Each S.Teacher do TotalTeachers:=TotalTeachers + 1
          EndFor
       EndFor
   EndFor
ndFor
Minister ←MinistryOfEducation(TotalTeachers)
```

Example - Query after Evolution

```
Ministry Of Education ← Minister (How many teachers do we have?)
      TotalTeachers : = 0
      P :- MinistryOfEducation . ListOfUniversties
For  Each P.University do University←MinistryOfEducation (How many teachers do you have?)
      Q :- University.ListOfFaculties
For Each Q.Faculty do Faculty ← University (How many teachers do you have?)
      R :- Faculty.ListOfSchools
 For Each R.School do School ← Faculty (How many teachers do you have?)
     S :- School.ListOfDepartments
 For Each S.Department do Department  ← School (How many teachers do you have?)
     T:-Department.ListOfTeachers
For Each T.Teacher do TotalTeachers:=TotalTeachers + 1
          EndFor EndFor
   EndFor
EndFor  EndFor
Minister ←MinistryOfEducation(TotalTeachers)
```

4. Adaptive Programming

- The object-oriented technology encapsulates data and functionality in classes. The implementation has been shielded for changing data structures and all changes concern only the interface.
- Some applications suffer under frequently changes in class hierarchy.
- Adaptive programming encapsulates class hierarchies using traversal strategies and visitors.
- Adaptive programming enables the application to have an interface to the class hierarchy, i.e. the application is not sensitive to the changes in the underlying class hierarchy.

Adaptive Programming

- The sequence of messages in operations depends on the class hierarchy.
- Should the class hierarchy be changed because of the evolution, usually also the sequence of messages must be changed, because some navigation paths have changed.
- Very often only the operation of massage propagation should be inserted or deleted. The goal is they can be generated automatically.

Adaptive Programming

- In the given example, only the leaf-class contains data that contribute in a nontrivial way to the result. This class has a method that contains some nontrivial semantics which must be explicitly described (wrappers in language DEMETER).
- All other classes propagate the message to their successors and bodies of their corresponding methods contain only the trivial semantics of propagation. This can be generated automatically using the current class hierarchy.

Example in DEMETER

```
*operation*  // operational clause
  void accumulateTeachers(int& totalTeachers)
  *traverse* // traversal clause
  *from* MinistryOfEducation
  *to*     Teacher
  *wrapper* Teacher // behavioral section
  *prefix*
  (@ totalTeachers = totalTeachers + 1; @)
```

Encapsulated class hierarchy

What has to be done when reaching nodes

Example Explained

- The program in DemeterJ (a propagation pattern) says that all paths from the root-class MinistryOfEducation to the leaf-class Teacher should be used (no constraints).
- Every time an object of the class Teacher will be reached, the counter totalTeacher will be increased.
- The segment between symbols @ will be embedded in the body of the generated method for class Teacher. In bodies of other methods, only the propagation code will be placed.

Language DEMETER

- The programming language DEMETER supports the idea of adaptive programming.
- A program in DEMETER will be translated into C++ or Java.

5. The Law of Demeter

- To get the possibility of using adaptive programming we have to accept some constraints for calling methods, i.e. constraints concerning the goal object, which we can send a message to.
- These constraints are described by the Law of Demeter
  - It is a style rule for designing object-oriented systems (Ian Holland, 1987)
  - General form: “Each unit should have only limited knowledge about other units. Only units “closely” related to the current unit should be talked to.”

The Law of Demeter

- Law of Demeter – main idea
  - Old Greek wisdom: “Don’t talk to strangers.”
- Each unit should

  “Only talk to your immediate friends”
- When writing a method, one should not hardwire the details of the class structure into that method
- We need a form of the Law of Demeter, which would be efficiently computable

  => class form of the Law of Demeter defines

  what does it mean “closely related”

The Law of Demeter – Class Form

- Closely related methods to method f:
  - other methods of class of `this/self` of f (inheritance) and other argument classes of f
  - methods of immediate part classes of class of f (aggregation)
    - stored aggregation => data members
    - computed aggregation

      => classes that are return types

      of methods of class of `this/self` ()

The Law of Demeter – Object Form

- However, it is the Object Form of the Law of Demeter which expresses the style rule we really want.
- Unfortunately, whether a program satisfies the object form in undecidable at compile-time.
- However, we can run the program and check at run-time whether violations occur.
- Rumbaugh summarizes the Law of Demeter as:

  A method should have limited knowledge

  of an object model.

The Law of Demeter

- To understand the meaning of one class, you need not understand the details of many other classes.
- A method must be able to traverse links to obtain its neighbors and must be able to call operations on them, but

  it should not traverse a second link from the neighbor to a third class.

- Ideal class graph: all classes participating in the operation are friends, even “far” away classes.

6. Class Dictionary

- A **class dictionary** defines the class structure of an application; it defines the application's classes and the relationships between them.
- In particular, it describes the inheritance hierarchy *and* the reference hierarchy of classes. The reference hierarchy consists of all binary relationships other than the inheritance (is-a) relationship. This includes the aggregation (part-of) and uses relationships.
- The Demeter tool will automatically take care of Java class definitions. This results in higher flexibility when changes occur.

Adaptive Design

- Key idea:
  - Introduce an ideal class graph
  - Write current behavior in terms of ideal class graph

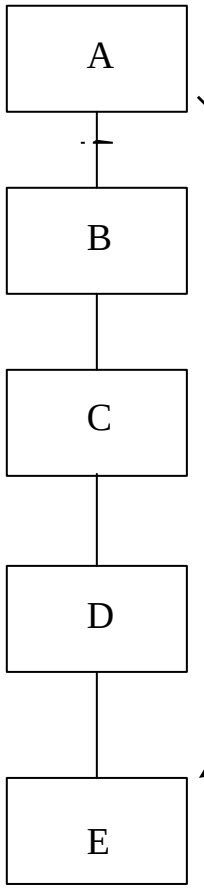
    => “as if all were friends”
  - Map ideal class graph flexibly into concrete class graph using traversal strategies

    => concatenation of methods (fixed navigation)

    has to be substituted by

    traversal strategies (adaptive navigation)

Adaptive Design



Dataflow and
Control flow

Closely
Related to A

```
Class A {public: B b;};
Class B {public: C c;};
Class C {public: D d;};
Class D {public: E e;};
Class E {public: void Goal();};
```

Class E is not a friend
=> a fixed navigation:

violates the Law of Demeter

{this . b.c.d.e. Goal()}

=> it has to be substituted by

=> an adaptive navigation:

{traversal.CG(this,“from A to E“).Goal()}

Description of the class graph

What if your friends are far away?

- You send an agent to them to collect the information you need.
  - Approximate Directions (adaptive design):

    You give directions about what kind of information to collect but you don't care about accidental details of the travel.
  - Detailed Directions (object-oriented design):

    You give detailed travel directions.

7. Traversal Strategies

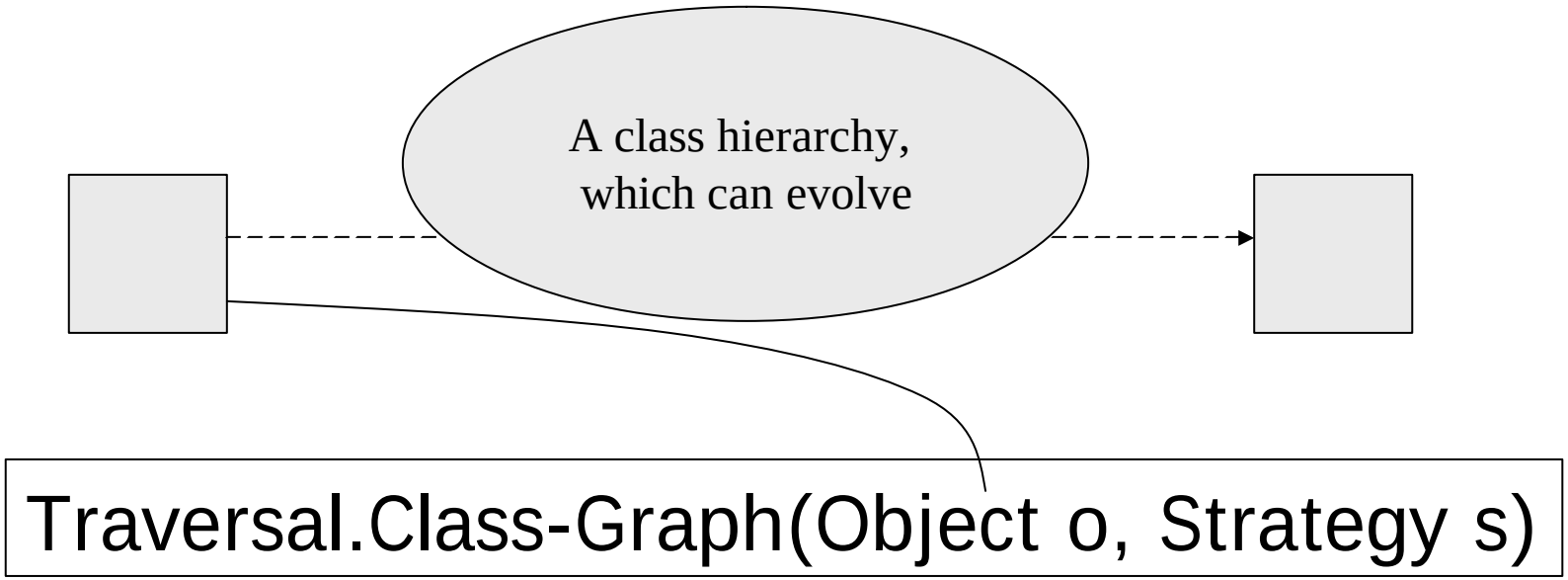
- **Traversal strategies** along with **visitor classes** define a specific behavior for a collection of classes.
- The behavior of the application as a whole is described by a set of traversal strategies and a set of visitor classes.
- The advantage of using this combination as opposed to regular Java code is their adaptiveness.

Traversal Strategies

- Traversals with visitors help to decouple class structure from the behavior of a program by not hard-wiring the details of the class structure into the program.
- Consequently, the program's behavior is less prone to change when the class structure is modified.
- In addition, programs written with traversals and visitors are more concise since trivial structure traversal code does not need to be specified and is again left to the tool's code generator.

Traversal strategies create friends

- Class Traversal is an intermediate class between classes that need to communicate



8. Propagation Pattern

- Propagation pattern (adaptive method) encapsulates the behavior of an operation into one place, thus avoiding the scattering problem, but also abstracts over the class structure, thus avoiding the tangling problem as well.
- Behavior is expressed as a high-level description of how to reach the participants of the computation (called a traversal strategy), plus what to do when each participant has been reached (called an adaptive visitor)

9. Visitor

- Actions to be performed:
  - Generate a subgraph of the Class Dictionary Graph
    - Specified by constraints
      in the adaptive program
  - Attach a method to each vertex in the subgraph
    - Method signature given by the adaptive program
  - Add a code to each generated method
    - Primary class gets “Visitor” code, while others get a call to children in subtree

Conclusion

- The main characteristic of adaptive programs is that they define behavior without specifying the detailed structure of the chain of participating objects.
- Therefore, programs are less dependent on a specific class structure.
- Therefore, programs become more adaptive to changes and evolution.

Benefits of Adaptive Programming

- Robustness to changes
- Shorter programs
- Design matches program (for an operation)
- More understandable code
- Partially automated evolution
- Keep all benefits of OO technology
- Improved productivity