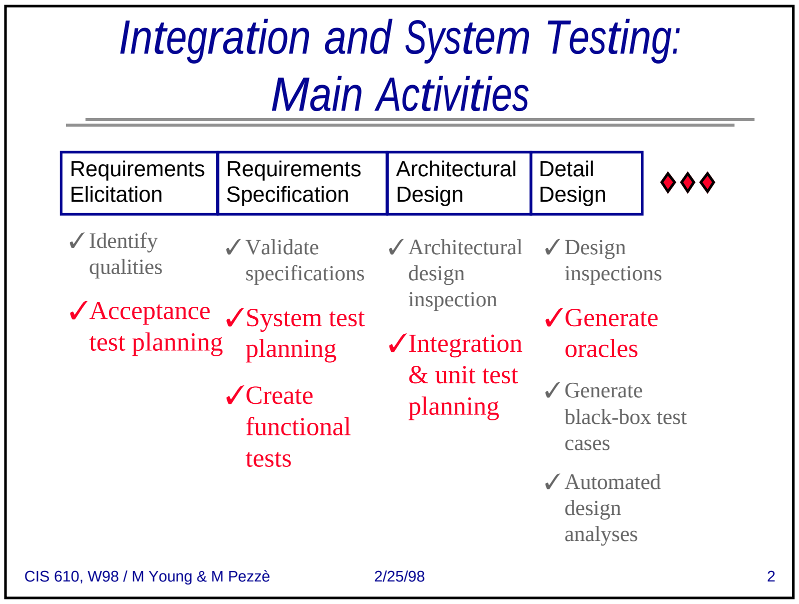
Integration & System Testing



Integration and System Testing: Main Activities

Requirements Elicitation	Requirements Specification	Architectural Design	Detail Design	♦♦♦
✓ Identify qualities ✓ Acceptance test planning	✓ Validate specifications ✓ System test planning ✓ Create functional tests	✓ Architectural design inspection ✓ Integration & unit test planning	✓ Design inspections ✓ Generate oracles ✓ Generate black-box test cases ✓ Automated design analyses	

CIS 610, W98 / M Young & M Pezzè 2/25/98 2

Integration and system testing comprises several steps during different development phases:

Acceptance test planning: In this step, that is part of requirements elicitation, the software engineers define the strategies for acceptance testing and identify the criteria to accept the final product.

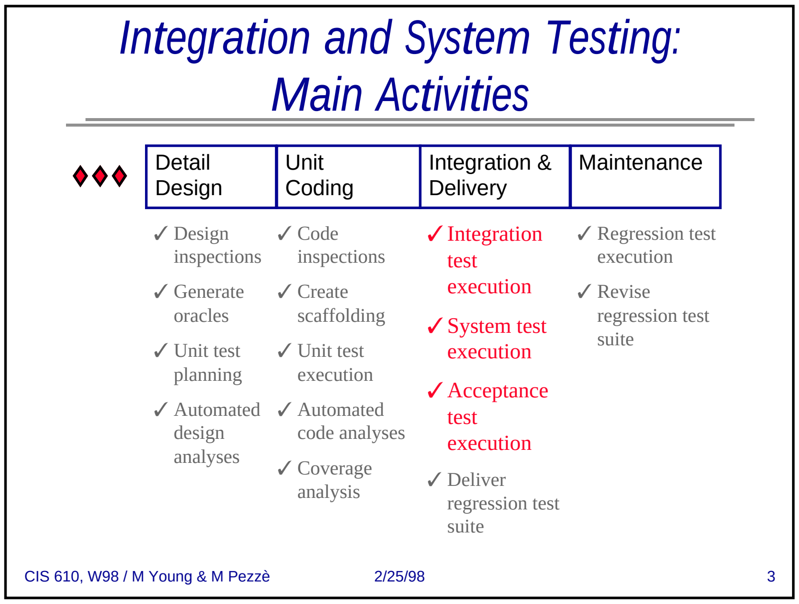
System test planning: In this step, that is part of requirements specification, the software engineers define the strategies for systems testing.

Create functional tests: requirements specifications are mapped to functional test cases, to be used during system testing.

Integration and unit test planning: definition of goals and process for integration testing. Integration testing and design strategies are defined and mutually related.

Generate oracles: Detailed design specifications are used to produce testing oracles to be used during unit testing.

•



Integration and System Testing: Main Activities

Detail Design	Unit Coding	Integration & Delivery	Maintenance
✓ Design inspections	✓ Code inspections	✓ Integration test execution	✓ Regression test execution
✓ Generate oracles	✓ Create scaffolding	✓ System test execution	✓ Revise regression test suite
✓ Unit test planning	✓ Unit test execution	✓ Acceptance test execution	
✓ Automated design analyses	✓ Automated code analyses	✓ Deliver regression test suite	
	✓ Coverage analysis		

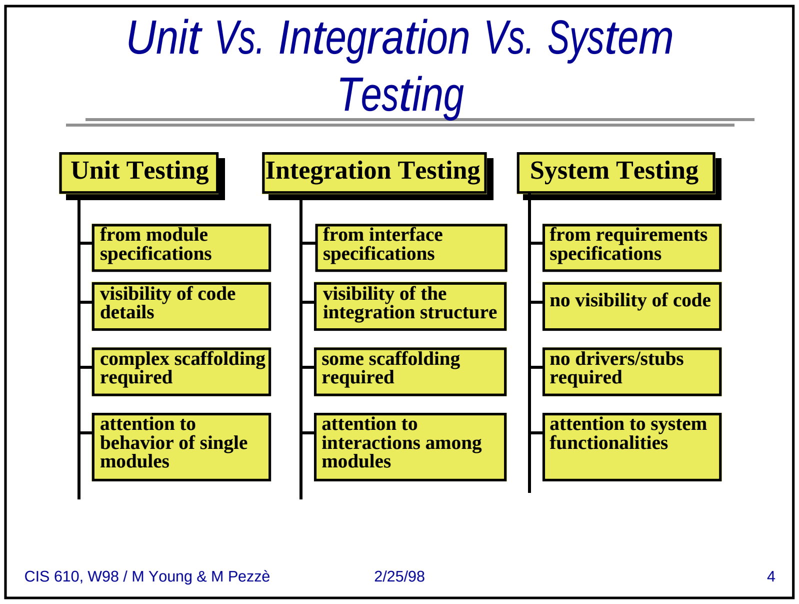
CIS 610, W98 / M Young & M Pezzè 2/25/98 3

(cont......)

Integration test execution: During integration and delivery, integration tests are defined and executed according to the plans defined during the architectural design. Communication and interface errors are identified and corrected.

System test execution: Functional tests defined during requirements specifications are executed. The functionalities of the system as a whole are exercised according to the requirements specifications.

Acceptance test execution:Tests defined as part requirements elicitation are executed. The functionalities of the whole system are tested with respect to the expectations of the final users.



Unit Vs. Integration Vs. System Testing

Unit Testing
from module specifications
visibility of code details
complex scaffolding required
attention to behavior of single modules
Integration Testing
from interface specifications
visibility of the integration structure
some scaffolding required
attention to interactions among modules
System Testing
from requirements specifications
no visibility of code
no drivers/stubs required
attention to system functionalities

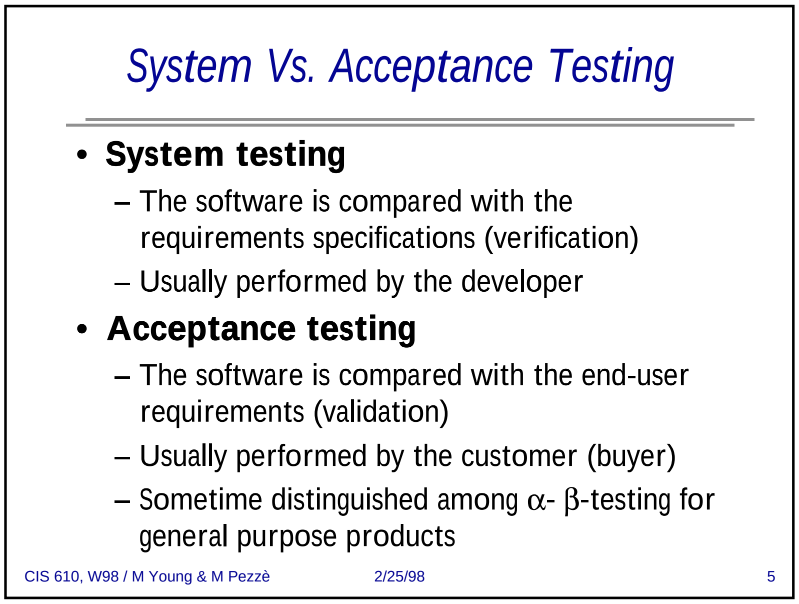
CIS 610, W98 / M Young & M Pezzè 2/25/98 4

Unit, integration, and system testing are complementary activities with different goals and execution procedures.

Unit testing focuses of the behavior of small units. Tests can be derived from module specifications or source code. Complex scaffoldings to set up the environment and check the results are usually required.

Integration testing focuses on communication and interface problems that may arise during module integration. Tests can be derived from module interfaces and detailed architecture specifications. some scaffolding is required, usually derivable from unit testing scaffoldings.

System testing focuses on the behavior of the system as a whole. Tests are derived from requirements specifications; code is seen as a black box. The system can be executed without the support of scaffoldings (a partial exception is embedded code, where some simulation of the embedding environment may be required).



System Vs. Acceptance Testing

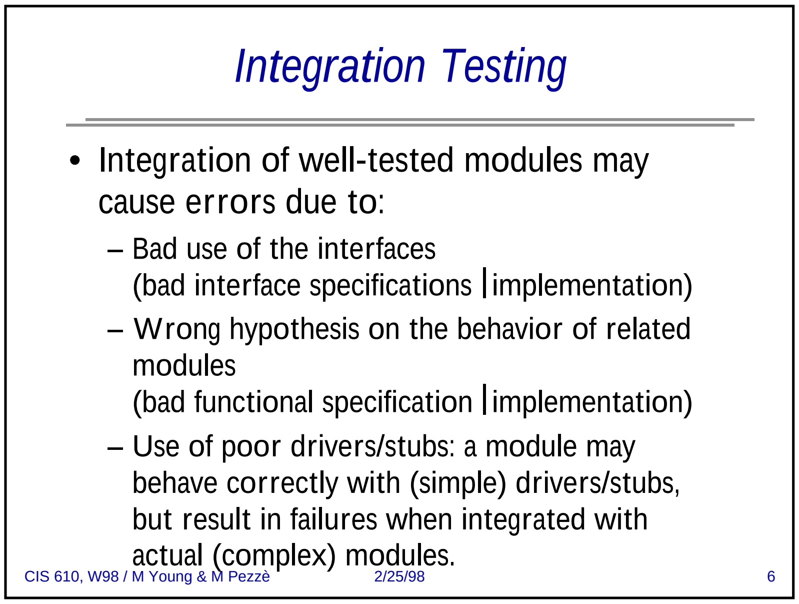
- **System testing**
  - The software is compared with the requirements specifications (verification)
  - Usually performed by the developer
- **Acceptance testing**
  - The software is compared with the end-user requirements (validation)
  - Usually performed by the customer (buyer)
  - Sometime distinguished among α- β-testing for general purpose products

CIS 610, W98 / M Young & M Pezzè 2/25/98 5

Both system and acceptance testing focuses on the whole system, but they are performed in different ways with different goals.

System testing is performed by the developers who have large visibility (and knowledge) of the structure of the system, but are not final users of the system itself; acceptance testing is performed by the final users who know very little about the structure of the system and its details, but are perfectly aware of the actual requirements of the operative environment.

System testing is mostly based on requirements specifications, i.e., the viewpoint of the developers and aims at verifying the system, i.e., check is the system has been built correctly (according to the requirements specifications). Acceptance testing is mostly based on the feeling of the final users, and aims at validating the system, i.e., check if the system is correct, by meeting the expectations of the final users.



Integration Testing

- Integration of well-tested modules may cause errors due to:
  - Bad use of the interfaces (bad interface specifications | implementation)
  - Wrong hypothesis on the behavior of related modules (bad functional specification | implementation)
  - Use of poor drivers/stubs: a module may behave correctly with (simple) drivers/stubs, but result in failures when integrated with actual (complex) modules.

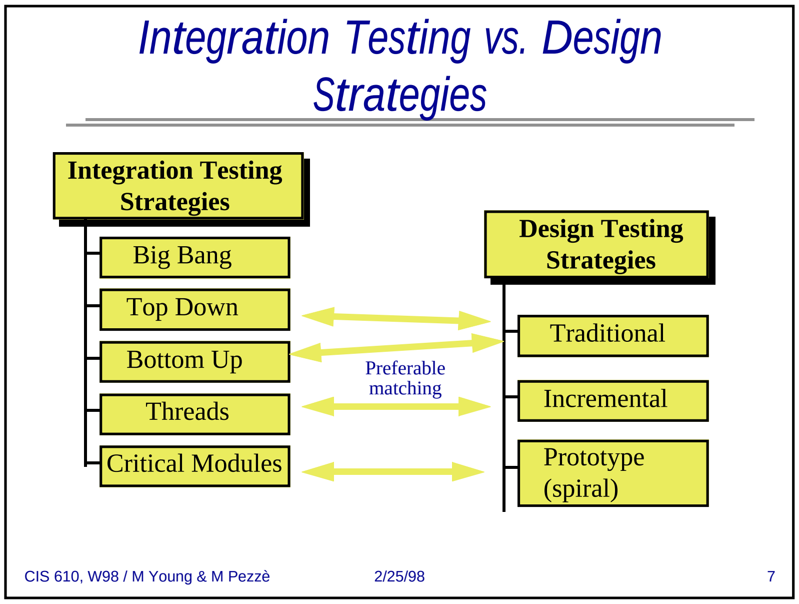
CIS 610, W98 / M Young & M Pezzè 2/25/98 6

Well tested (even correct) units may still cause problems when integrated due to:

bad use of interfaces, when units violate hypothesis on the interfaces of used modules

wrong hypothesis on the behavior of the used components, that results in bad use of returned value.

use of poor scaffolding. Scaffoldings approximate the “embedding”. When the quality of scaffoldings is reduced to contain costs, their behavior may be a bad approximation of the embedding and thus the module may behave differently than expected when “embedded” in the final system.



Integration Testing vs. Design Strategies

CIS 610, W98 / M Young & M Pezzè 2/25/98 7

Integration testing can be based on different strategies:

big bang: all units are put together and tested. This strategy does not require drivers and stubs, but makes the job of the software engineers very hard, because faults can hardly be related to subset of modules.

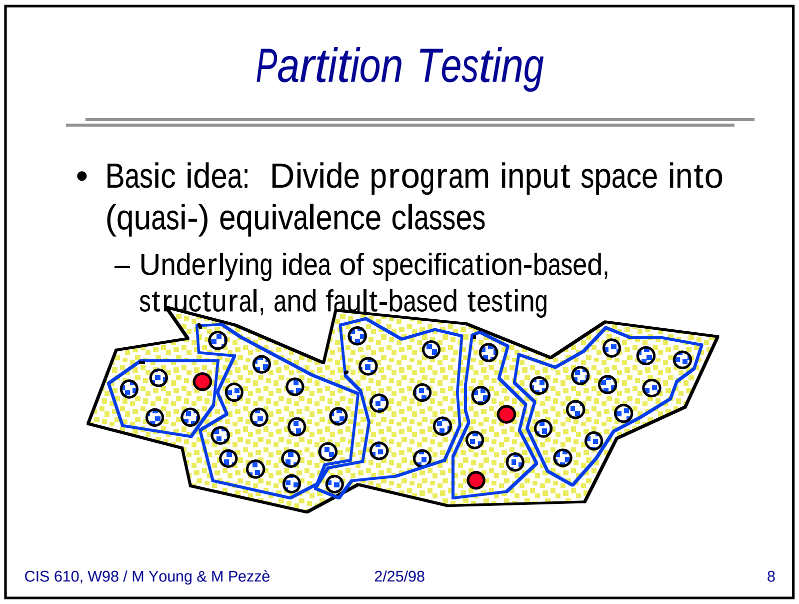
top down: units are merged starting from the more general to the more detailed ones. This strategies requires complex stubs, but does not require drivers.

bottom up: units are merged starting from the more detailed to the more general ones. This strategies requires complex drivers, but does not require stubs.

threads: units are merged according to expected execution threads

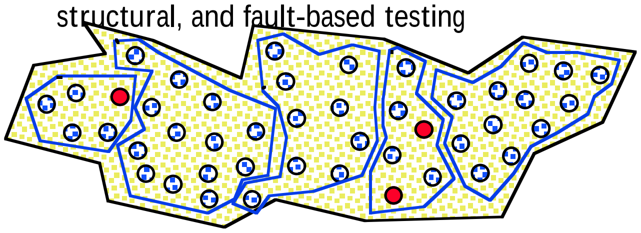
critical modules: units are merged according to their criticality level.

Although integration testing and design strategies are not strictly related, careful choices of them can optimize effort: top-down and bottom-up integration testing strategies are more suited to traditional design strategies; threads integration testing strategies are more suited to incremental design; critical module integration testing strategies are more suited to prototype based approaches (e.g., the spiral model).



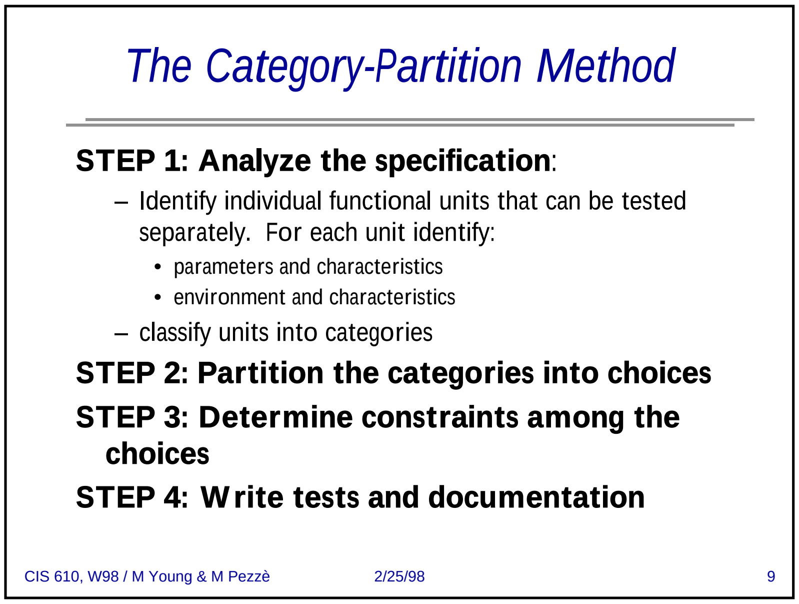
Partition Testing

- Basic idea: Divide program input space into (quasi-) equivalence classes
  - Underlying idea of specification-based, structural, and fault-based testing



CIS 610, W98 / M Young & M Pezzè 2/25/98 8

System testing approaches are partition strategies: the input domain is partitioned in equivalence classes from which tests are derived. Specific system testing strategies differ in the way the input domain is partitioned and in the way tests are derived from partitions. In this tutorial, we illustrate the principle by referring to the Category-partition method proposed by Ostrand and Balcer.



The Category-Partition Method

STEP 1: Analyze the specification:

- Identify individual functional units that can be tested separately. For each unit identify:
  - parameters and characteristics
  - environment and characteristics
- classify units into categories

STEP 2: Partition the categories into choices

STEP 3: Determine constraints among the choices

STEP 4: Write tests and documentation

CIS 610, W98 / M Young & M Pezzè 2/25/98 9

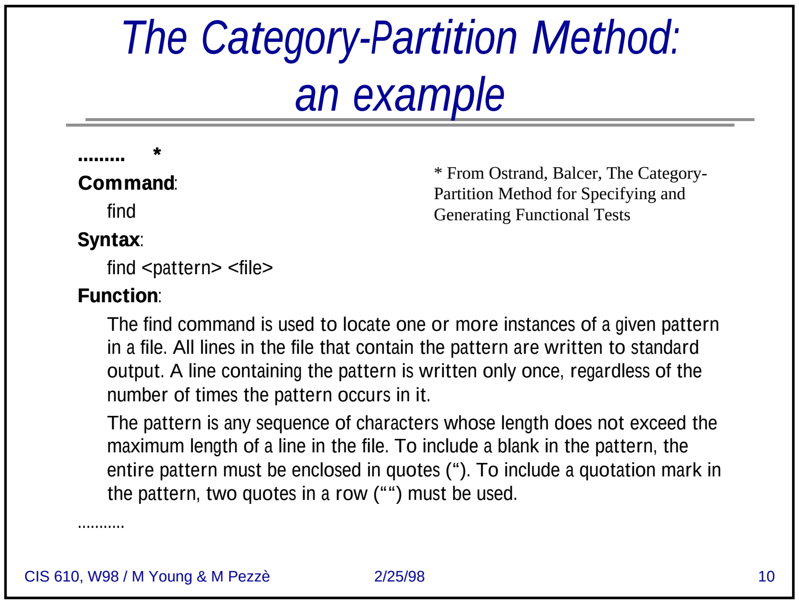
The Category-partition method is based on four main steps:

a detailed analysis of the (informal) specifications that aims at identifying individual functionalities (functional units) that can be tested separately. Each functional unit is described by giving the calling environment (parameters and non-local variables). Functional units are then classified into categories.

categories are partitioned into choices, that identify different sets of values for each element of the calling environment

constraints are added to reduce the number of choices by eliminating trivial ones.

tests and test documentation is finally produced.



The Category-Partition Method: an example

.... *

Command:

find

Syntax:

find <pattern> <file>

Function:

The find command is used to locate one or more instances of a given pattern in a file. All lines in the file that contain the pattern are written to standard output. A line containing the pattern is written only once, regardless of the number of times the pattern occurs in it.

The pattern is any sequence of characters whose length does not exceed the maximum length of a line in the file. To include a blank in the pattern, the entire pattern must be enclosed in quotes ("). To include a quotation mark in the pattern, two quotes in a row ("") must be used.

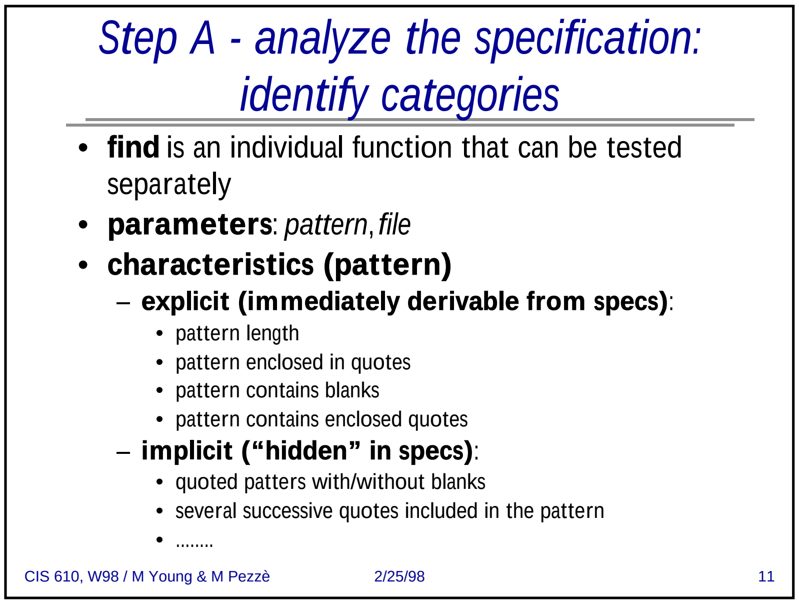
....

* From Ostrand, Balcer, The Category-Partition Method for Specifying and Generating Functional Tests

CIS 610, W98 / M Young & M Pezzè 2/25/98 10

EXAMPLE:

This is an example of function unit identified as part of the first step of the category-partition method.



Step A - analyze the specification: identify categories

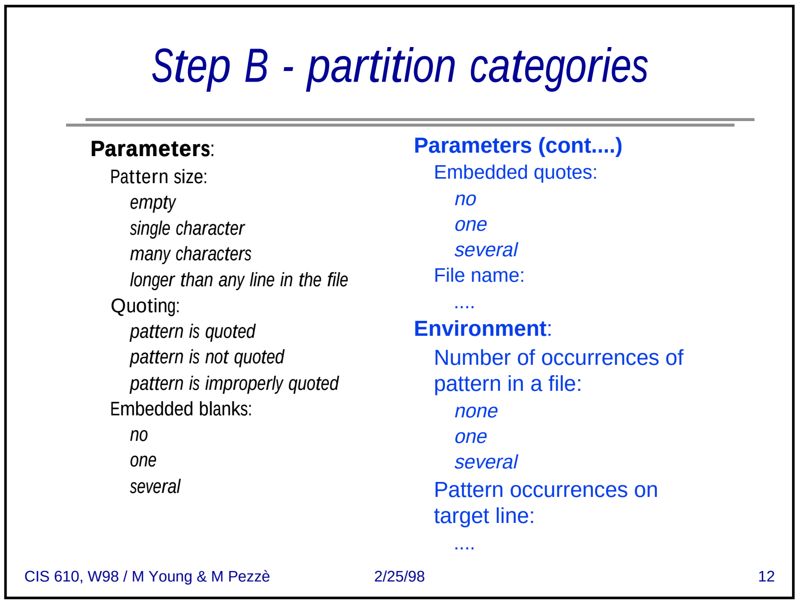
- **find** is an individual function that can be tested separately
- **parameters**: *pattern*, *file*
- **characteristics (pattern)**
  - **explicit (immediately derivable from specs)**:
    - pattern length
    - pattern enclosed in quotes
    - pattern contains blanks
    - pattern contains enclosed quotes
  - **implicit ("hidden" in specs)**:
    - quoted patters with/without blanks
    - several successive quotes included in the pattern
    - ....

CIS 610, W98 / M Young & M Pezzè 2/25/98 11

EXAMPLE:

once identified a functional unit, we must identify the calling environment and define categories. This activity can be based on the explicit content of the specification or on implicit assumptions, that are usually clear to the expert developers.

This step often helps in highlighting ambiguities and incompleteness in the original specifications.



Step B - partition categories

Parameters:
- Pattern size:
  - *empty*
  - *single character*
  - *many characters*
  - *longer than any line in the file*
- Quoting:
  - *pattern is quoted*
  - *pattern is not quoted*
  - *pattern is improperly quoted*
- Embedded blanks:
  - *no*
  - *one*
  - *several*

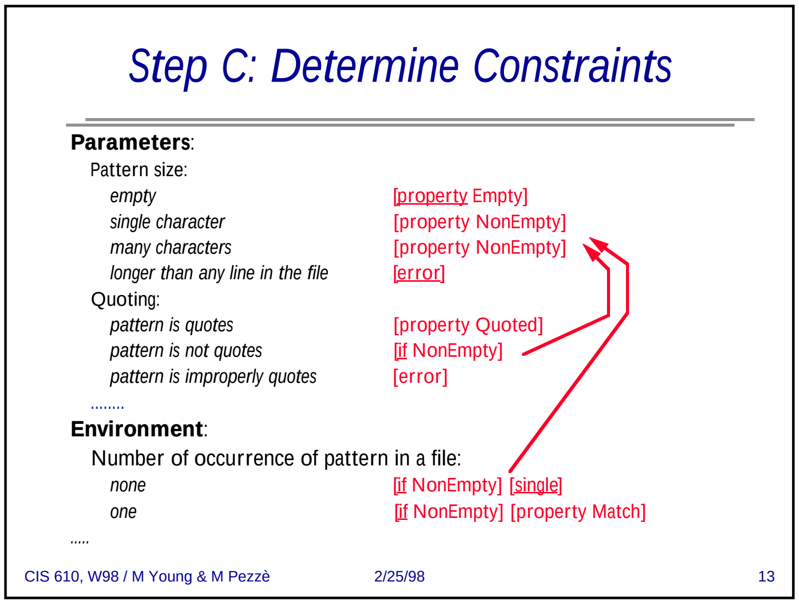
Parameters (cont....)
- Embedded quotes:
  - *no*
  - *one*
  - *several*
- File name:
  - ....

Environment:
- Number of occurrences of pattern in a file:
  - *none*
  - *one*
  - *several*
- Pattern occurrences on target line:
  - ....

CIS 610, W98 / M Young & M Pezzè 2/25/98 12

EXAMPLE:

categories are then partitioned into choices. For the chosen example, choices are the size of the pattern, the presence of quoting or embedded blanks, the presence of embedded quotes, the number of occurrences of the pattern in the file, etc.



Step C: Determine Constraints

Parameters:

Pattern size:

empty	[property Empty]
single character	[property NonEmpty]
many characters	[property NonEmpty]
longer than any line in the file	[error]

Quoting:

pattern is quotes	[property Quoted]
pattern is not quotes	[if NonEmpty]
pattern is improperly quotes	[error]

....

Environment:

Number of occurrence of pattern in a file:

none	[if NonEmpty] [single]
one	[if NonEmpty] [property Match]

.....

CIS 610, W98 / M Young & M Pezzè 2/25/98 13

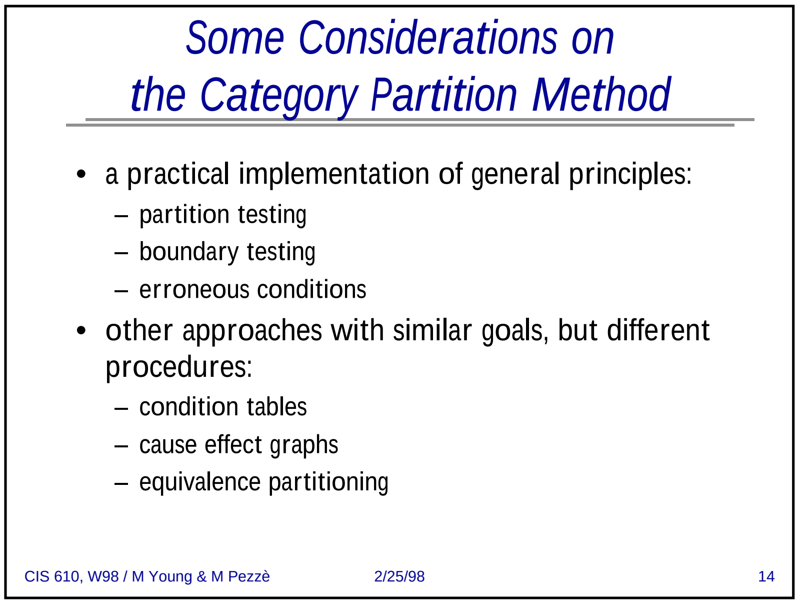
The simple choice of one item for each partition can generate too many test cases, most of which useless. For example, if the pattern is empty, the occurrences of quotes in the patterns are meaningless. In this step, partitions are annotated with keywords:

[property <name>] defines a name for a property. For example, *Pattern size: single character* is named as *NonEmpty*.

[if <property name>] indicates the conditions for choosing the property. For example, the choice of a *non quoted pattern* is meaningful only for properties of type *NonEmpty*

[error] indicates that the corresponding test is an erroneous input, and thus does not have to be combined with other choices. For example, a pattern longer than any line in the file is an erroneous input.

[single] indicates that the property shall be exercised in only one combination.



Some Considerations on the Category Partition Method

- a practical implementation of general principles:
  - partition testing
  - boundary testing
  - erroneous conditions
- other approaches with similar goals, but different procedures:
  - condition tables
  - cause effect graphs
  - equivalence partitioning

CIS 610, W98 / M Young & M Pezzè 2/25/98 14

The category partition methods provides a methodology to define system tests. Its efficacy relies on the ability of identifying function units, categories and partitions. The presented example suggests that categories and partitions must cover all possible combinations of input data, must take care of boundary cases (e.g., empty pattern), and must consider erroneous conditions (e.g., pattern longer than any line in the file). The method itself does not require such cases to be covered, but a good implementation will satisfy the following important principles:

partition testing: identify an “even” partition of the input domain

boundary testing: check all boundary conditions

erroneous condition testing: check erroneous inputs

Constraints are important for optimization and feasibility of derived tests, but not for the selection of the tests themselves.

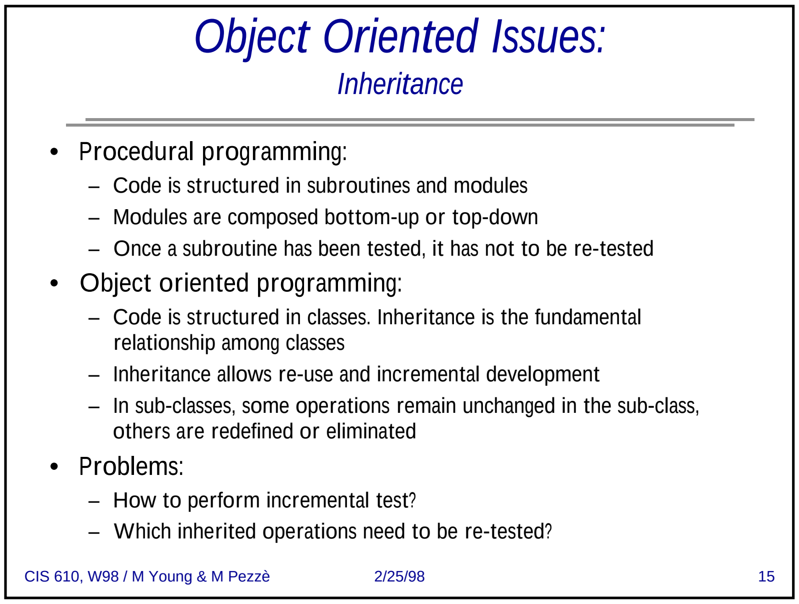
The same principles are implemented by other methods, that substitute categories and constraints with

condition tables

cause effect graphs

equivalence partitioning

Main difference among these methods is the complexity of the approach.



Object Oriented Issues:

Inheritance

- Procedural programming:
  - Code is structured in subroutines and modules
  - Modules are composed bottom-up or top-down
  - Once a subroutine has been tested, it has not to be re-tested
- Object oriented programming:
  - Code is structured in classes. Inheritance is the fundamental relationship among classes
  - Inheritance allows re-use and incremental development
  - In sub-classes, some operations remain unchanged in the sub-class, others are redefined or eliminated
- Problems:
  - How to perform incremental test?
  - Which inherited operations need to be re-tested?

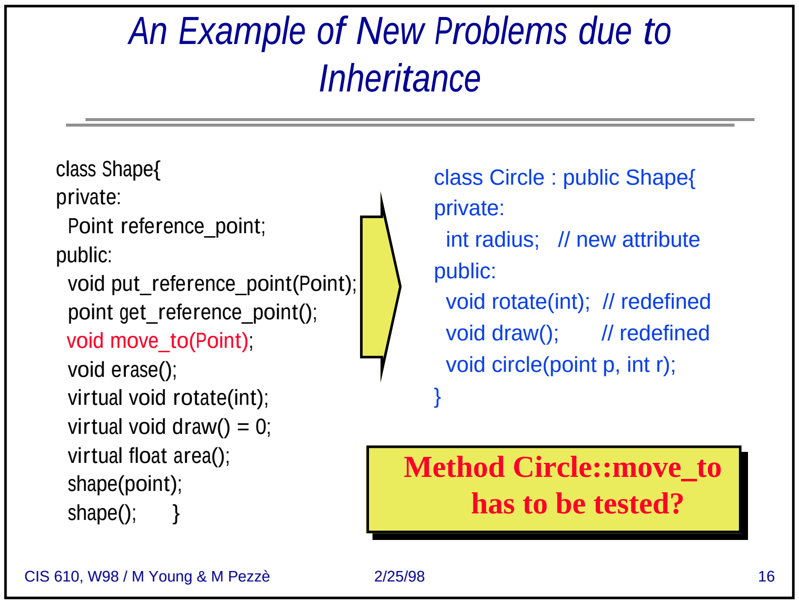
CIS 610, W98 / M Young & M Pezzè 2/25/98 15

System testing is based on requirements specifications, and it is thus independent from the specific design and implementation paradigm. Different design and implementation paradigms can impact integration testing.

In particular, object oriented software presents new problems and challenges.

First problem derives from inheritance. Procedural code is structured in functions and subroutines that are composed top-down or bottom-up. Once a subroutine has been tested and integrated, it has not to be retested. Object oriented programs are structured in classes. Inheritance among classes allows re-use and incremental development. Sub-classes enrich ancestors by adding, deleting or modifying state variables and methods.

The trivial approach of re-testing each class from scratch is a waste of time and resources. Efficient incremental testing of class hierarchies must be able to reduce the amount of test cases to be re-executed or re-designed from scratch without reducing the confidence level of testing.



An Example of New Problems due to Inheritance

```
class Shape{
private:
  Point reference_point;
public:
  void put_reference_point(Point);
  point get_reference_point();
  void move_to(Point);
  void erase();
  virtual void rotate(int);
  virtual void draw() = 0;
  virtual float area();
  shape(point);
  shape();     }
```

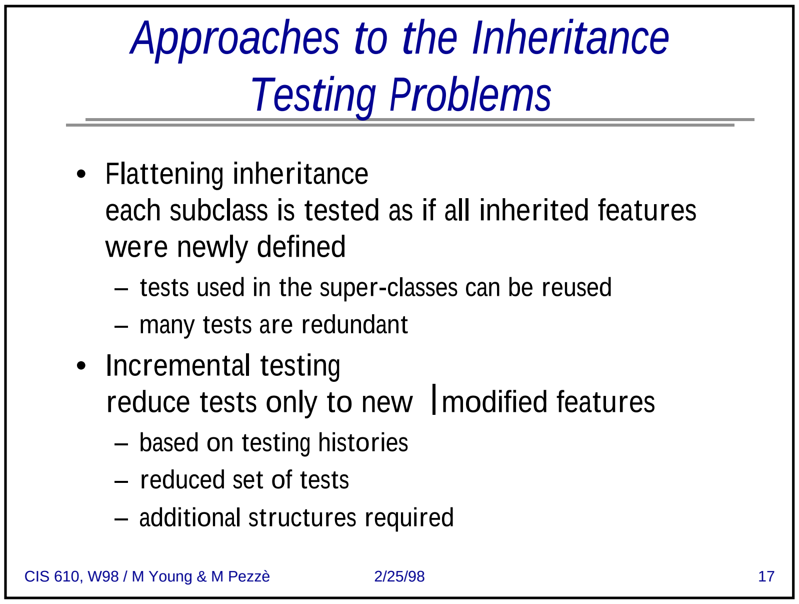
```
class Circle : public Shape{
private:
  int radius;   // new attribute
public:
  void rotate(int);  // redefined
  void draw();       // redefined
  void circle(point p, int r);
}
```

Method Circle::move_to has to be tested?

CIS 610, W98 / M Young & M Pezzè 2/25/98 16

EXAMPLE:

Class *Circle* inherits method *move_to* from class *Shape*. How can we determine if method *move-to* already tested for class *Shape* must be re-tested for class *Circle*?



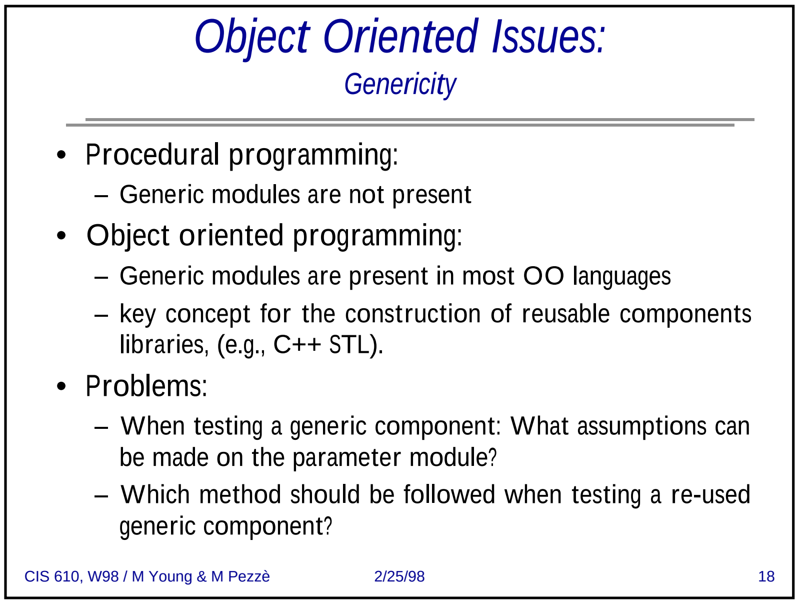
Approaches to the Inheritance Testing Problems

- Flattening inheritance
  each subclass is tested as if all inherited features were newly defined
  - tests used in the super-classes can be reused
  - many tests are redundant
- Incremental testing
  reduce tests only to new | modified features
  - based on testing histories
  - reduced set of tests
  - additional structures required

CIS 610, W98 / M Young & M Pezzè 2/25/98 17

A first set of approaches to the incremental testing of classes is the trivial one: each class is re-tested as if it were newly defined. As mentioned before such approaches are easily implementable referring to traditional testing techniques, but they can be extremely inefficient.

A second set of approaches try to reduce the amount of tests to be executed taking into account the tests already executed for the super classes. Such methods are based on concepts like testing histories, that allow to determine which test cases do not have to be re-executed, which must be re-execute, and which must be newly defined, depending on the relation of the class with its super-classes.

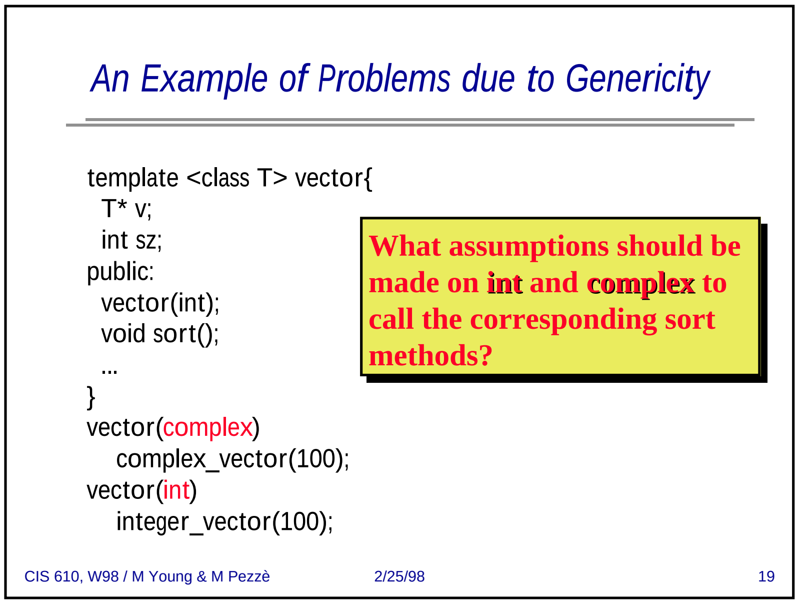

Object Oriented Issues:
Genericity

- Procedural programming:
  - Generic modules are not present
- Object oriented programming:
  - Generic modules are present in most OO languages
  - key concept for the construction of reusable components libraries, (e.g., C++ STL).
- Problems:
  - When testing a generic component: What assumptions can be made on the parameter module?
  - Which method should be followed when testing a re-used generic component?

CIS 610, W98 / M Young & M Pezzè 2/25/98 18

A second class of problems for object oriented programs is given by generic modules. Testing generic modules becomes even more difficult when dealing with libraries, where we have little information about the use that will be done of such classes.

Main problem of integration testing of generic modules is the identification of parameters that can assure a good coverage of the generic class.



An Example of Problems due to Genericity

```
template <class T> vector{
  T* v;
  int sz;
public:
  vector(int);
  void sort();
  ...
}
vector(complex)
    complex_vector(100);
vector(int)
    integer_vector(100);
```

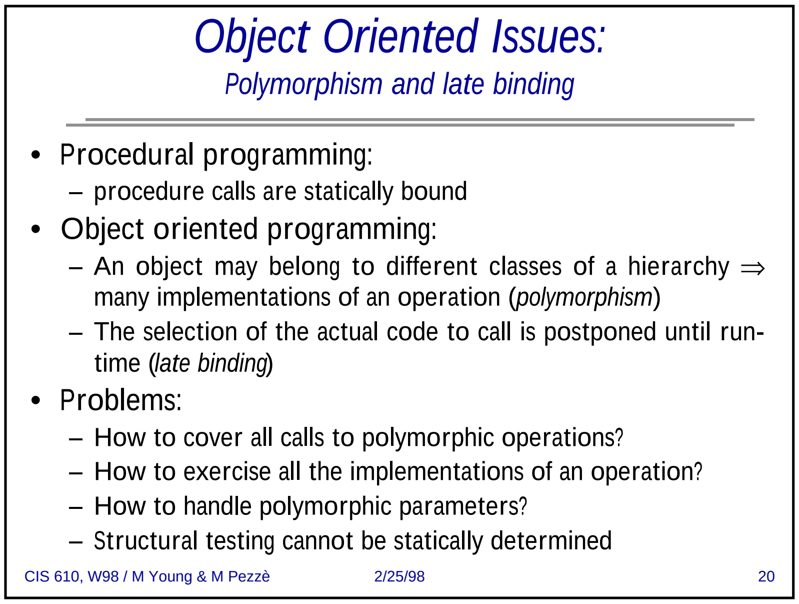
What assumptions should be made on int and complex to call the corresponding sort methods?

CIS 610, W98 / M Young & M Pezzè 2/25/98 19

EXAMPLE:

Class *vector* can be used with different parameters. Are test cases defined for parameter *int* enough for parameter *complex*, or do we need to re-test class *vector* with the new parameter. Is there a set of parameters that can ensure a good coverage of the testing of class *vector* before checking it in a library?

So far few solutions exists, mainly based on the restriction of the possible parameter, but no general solution has been thoroughly experimented so far.



Object Oriented Issues:

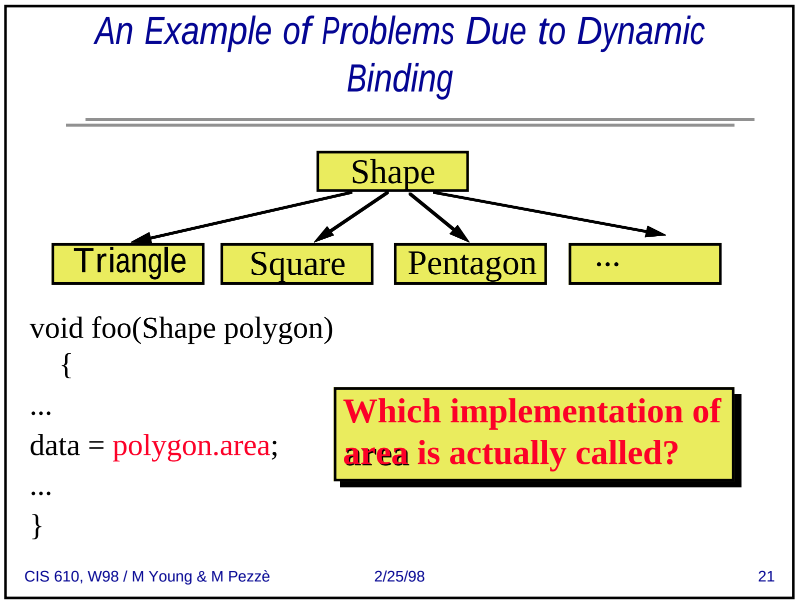
Polymorphism and late binding

- Procedural programming:
  - procedure calls are statically bound
- Object oriented programming:
  - An object may belong to different classes of a hierarchy ⇒ many implementations of an operation (*polymorphism*)
  - The selection of the actual code to call is postponed until run-time (*late binding*)
- Problems:
  - How to cover all calls to polymorphic operations?
  - How to exercise all the implementations of an operation?
  - How to handle polymorphic parameters?
  - Structural testing cannot be statically determined

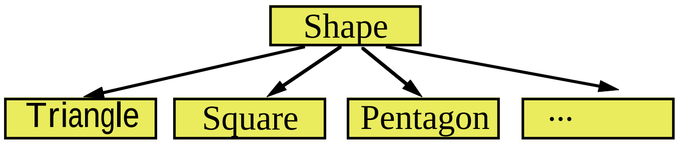
CIS 610, W98 / M Young & M Pezzè 2/25/98 20

Another class of problems of testing object oriented programs derives from polymorphism and late binding. In procedural programs, procedure calls are statically bound. In object oriented programs, objects can belong to different classes of the same hierarchy and the bindings between caller and called can be determined only at execution time.

Exhaustive testing in presence of dynamic binding presents new challenges, that cannot be approached with traditional testing techniques.



An Example of Problems Due to Dynamic Binding



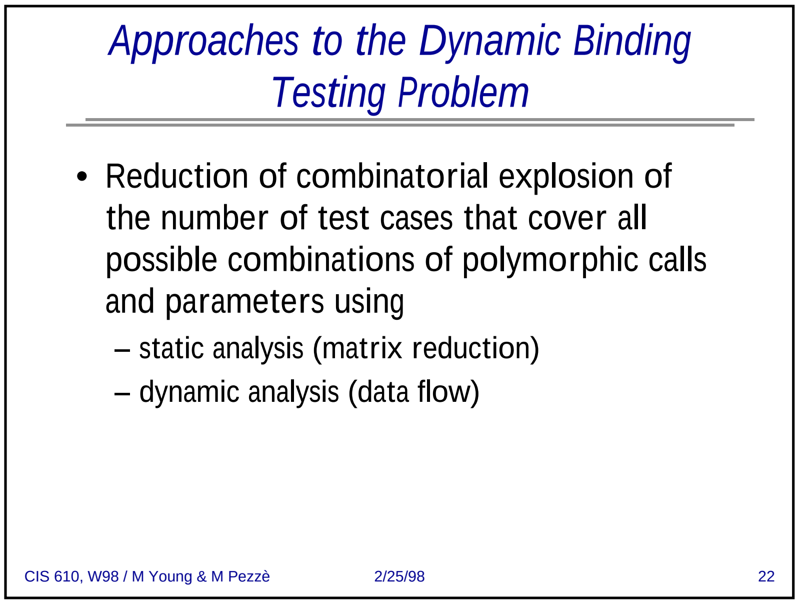
```
void foo(Shape polygon)
   {
...
data = polygon.area;
...
}
```

Which implementation of area is actually called?

CIS 610, W98 / M Young & M Pezzè 2/25/98 21

EXAMPLE:

Class *Shape* is specialized in different geometric figures. The call to method *area* can be dynamically bound to any object of any class in the hierarchy. The testing of the method call with an object of class *Triangle*, would not reveal faults due to the binding with the method of class *Square*.



Approaches to the Dynamic Binding Testing Problem

- Reduction of combinatorial explosion of the number of test cases that cover all possible combinations of polymorphic calls and parameters using
  – static analysis (matrix reduction)
  – dynamic analysis (data flow)

CIS 610, W98 / M Young & M Pezzè 2/25/98 22

Problems of testing in presence of dynamic biding are approached by reducing the combinatorial explosion of possible combination either statically, using matrix reduction techniques, or dynamically, using data flow analysis techniques.