M.Sc Computer Science

Object-Oriented Design and Programming

Revision

Oded Lachish

Email: oded@dcs.bbk.ac.uk

Web Page: http://www.dcs.bbk.ac.uk/~oded/OODP12/OODP2012.html

Question 1

(a) What is the motivation for using the observer design pattern

[marks: 4]

(b) Draw a UML class diagram of the observer design pattern.

[marks: 4]

(c) Describe a bug that can occur because of using the observer design pattern.

[marks: 4]

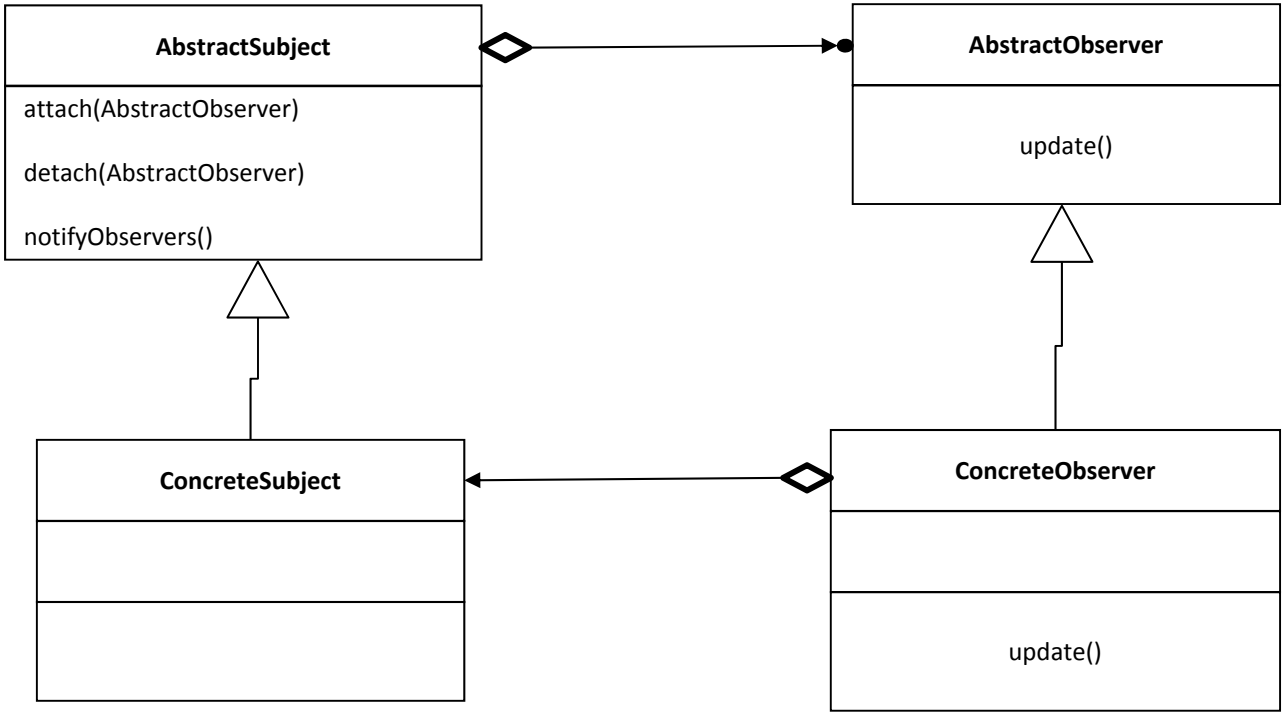
(d) Suggest a design pattern that can be combined with the observer design pattern explain why this combination makes sense.

[marks: 5]

(e) Draw the UML diagram of the combined design-patterns.

[marks: 5]

Observer – UML class diagram



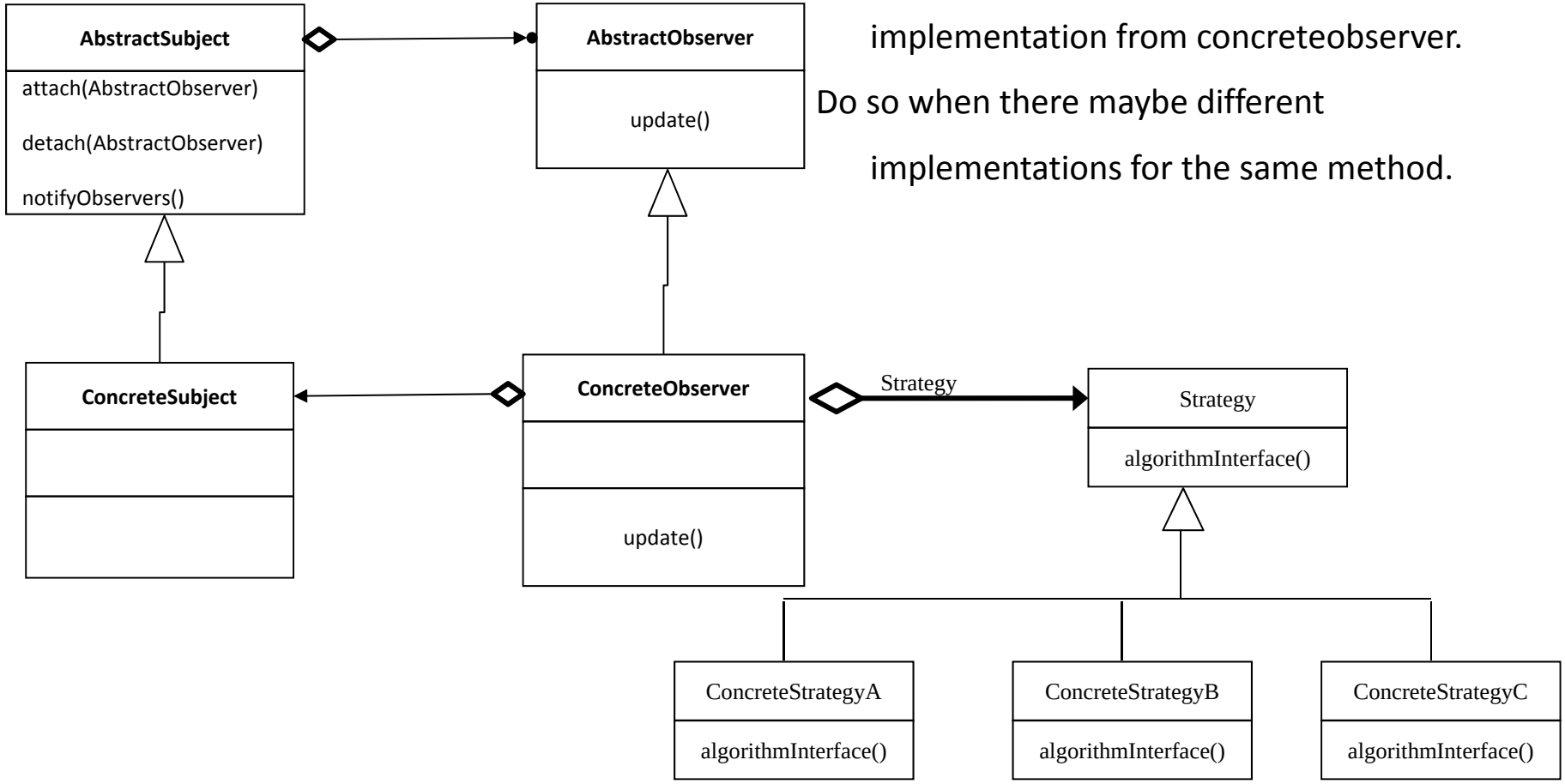
The observer is used when there are many objects (observers) that should be updated when a specific object (subject)changes its state. The observer design pattern gives a generic mechanism for implementing such behaviour when needed.

If one is not careful when using the observer pattern a “loop” may be created (need a bit extra explanation). This will result in a stack overflow error in run time.

Observer + Strategy

Use strategy in order to decouple the actual implementation from concreteobserver.

Do so when there maybe different implementations for the same method.



Question 2

(a) Draw a UML class diagram for the strategy design pattern.

[marks: 4]

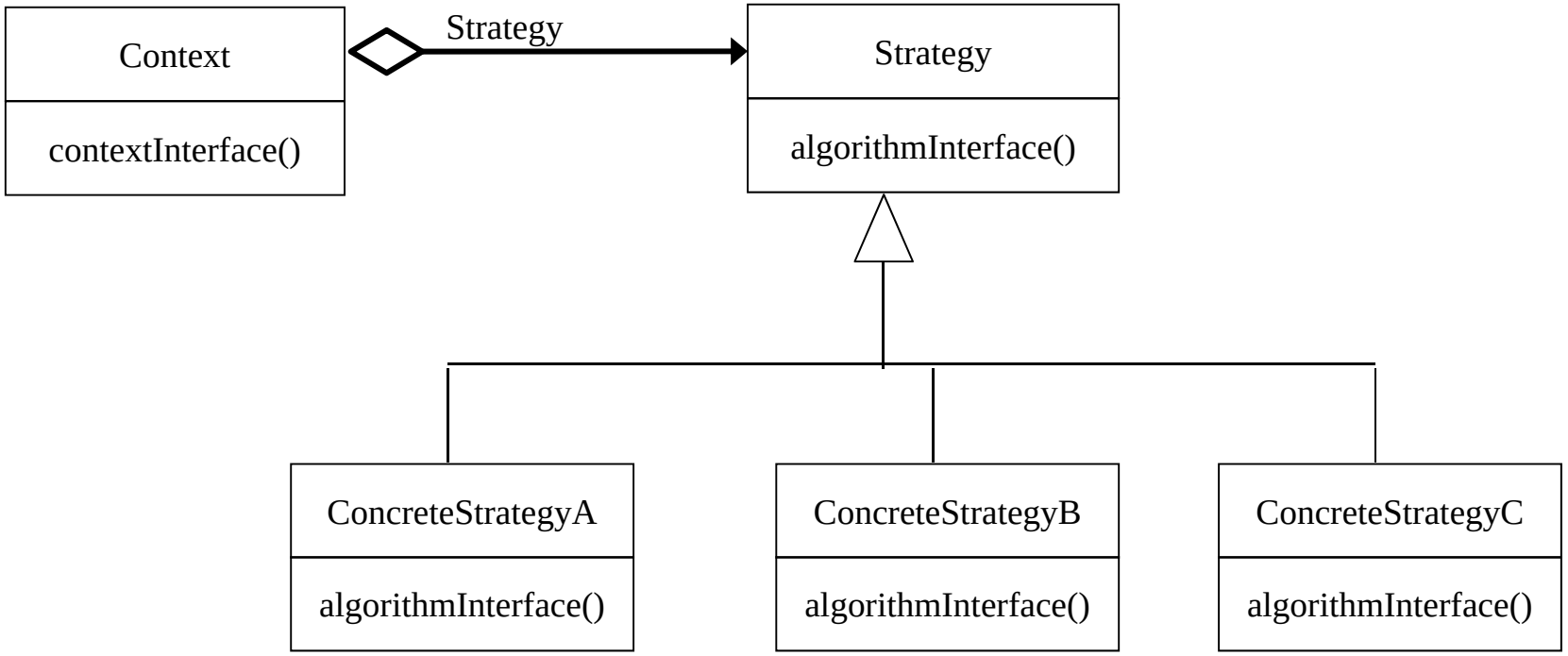
(b) Write example code where the strategy design pattern can be applied. Explain why the strategy design pattern may be refactored into your example.

[marks: 9]

(c) Write the resulting code after refactoring the strategy design pattern into the code you have previously given. Explain how the refactoring was done.

[marks: 13]

Strategy Pattern



Pre Strategy Code

```
public class Doer {

	private int iD;

	public Doer(int i) {
		iD = i;
	}

	public void doSomething() {
		switch(iD){
		case 1: nothing();
		break;
		case 2: talk();
		break;
		case 3: laugh();
		default: return;
		}
	}

	private void laugh() {
		System.out.println("Laugh");
	}

	private void talk() {
		System.out.println("Talk");
	}

	private void nothing(){
		System.out.println("Nothing");
	}
}
```

Pre Strategy Code

- Replace “int iD” with “Strategy doing”
- Use Eclipse IDE to create interface Strategy
- Use “source” option to create constructor with doing field
- Delete the old constructor
- Replace all the switch code with delegation “doing.doSomething()”
- Use Eclipse IDE to create method “doSomething” in interface doing
- Use Eclipse create a class “Laugh” implementing Strategy
- Copy the content of Laugh method into the doSomething method of the “Laugh strategy
- Repeat previous two steps for “Nothing” and “Talk”
- Remove redundant methods in Doer (laugh, talk, nothing)

(in reality also tests and other classes must be updated, but this is sufficient for out purpose)

Post Strategy Code

```
public class Doer {

	private Strategy doing;

	public Doer(Strategy doing) {
		this.doing = doing;
	}

	public void doSomething() {
		doing.doSomething();
	}
}
```

```
public interface Strategy {

	public void doSomething();

}
```

```
public class Laugh implements Strategy {

	@Override
	public void doSomething() {
		System.out.println("Laugh");
	}
}
```

```
public class Nothing implements Strategy {

	@Override
	public void doSomething() {
		System.out.println("Nothing");
	}
}
```

```
public class Talk implements Strategy {

	@Override
	public void doSomething() {
		System.out.println("Talk");
	}
}
```

Question 3

(a) What is the main problem with the ‘singleton’ design pattern?

[marks: 4]

(b) Write code for a class that is both a singleton and a factory method for three polymorphic classes

[marks: 4]

Singleton + Factory

```
public class Single {
      private static Single _single;
      private Single() { /* private constructor */ }
      public static synchronized Single getSingle() {
      if (_single == null) _single = new Single();
            return _single;
      }

      public Strategy getStrategy(String which)
      {
            if(which.equals ("Talk")){
                  return new Talk();
            }
            if(which.equals ("Nothing")){
                  return new Nothing();
            }
            if(which.equals ("Laugh")){
                  return new Laugh();
            }
            return null;
      }

}
```

Pattern Name: Indirection

- **The Singleton design pattern causes problems in testing.**

This happens since the fact that it adds coupling.

- GoF Design Patterns:
  - Strategy
  - Abstract Factory

Question 4

(a) Describe the indirection Grasp pattern give an example of a design-pattern that uses indirection. Explain why indirection is used in this design pattern.

[marks: 4]

(b) Describe the polymorphism Grasp pattern give an example of a design-pattern that uses indirection. Explain why indirection is used in this design pattern.

[marks: 4]

Pattern Name: Indirection

- **Problem It Solves:** Where to assign a responsibility, to avoid direct coupling between two or more things
- **Solution:** Assign the responsibility to an intermediate object

- GoF Design Patterns:
  - Strategy
  - Abstract Factory

Pattern Name: Polymorphism

- **Problem It Solves:** How to handle alternatives based on type? How to create pluggable software components?
- **Solution:** When related alternatives or behaviour vary by type, assign responsibilities for the behaviour using polymorphism

Alternatives based on type – replacing code by classes and types

Pluggable software components – viewing components in client server relationships, how can you replace one server component with another without affecting the client

- GoF Design Patterns: Strategy, Abstract Factory

Question 5

(a) Describe the test driven development process (TDD).

[marks: 8]

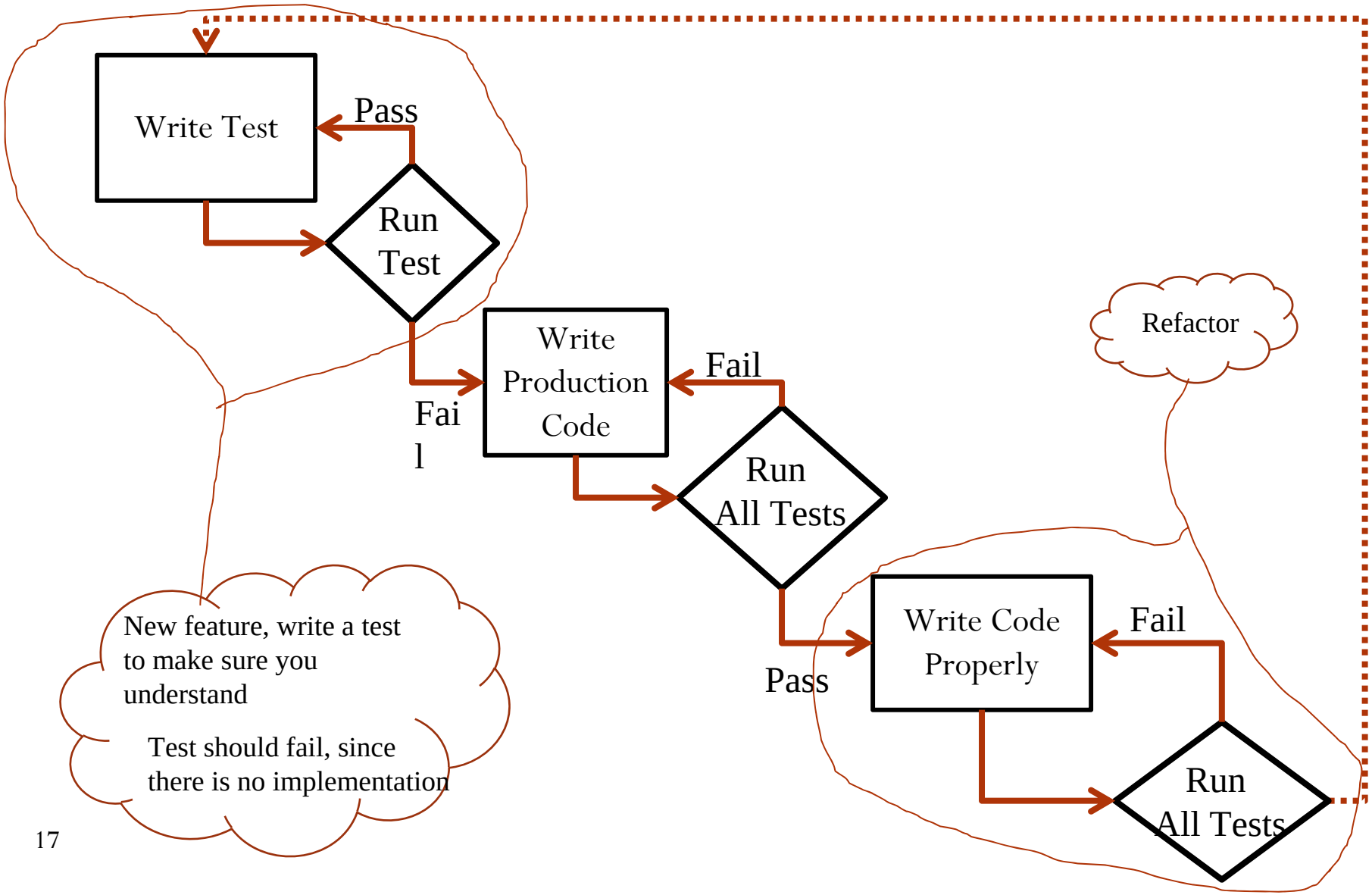
(a) What are the main advantages and disadvantages of the TDD.

[marks: 8]

(a) Name two tools that support TDD. Explain how these tools are used in order to support the TDD process.

[marks: 8]

TDD Cycle


Write Test
Pass
Run Test
Fail
Write Production Code
Fail
Run All Tests
Pass
Write Code Properly
Fail
Run All Tests
Refactor
New feature, write a test to make sure you understand
Test should fail, since there is no implementation

Test Driven Development (TDD)

Philosophy

- Untested code is “not code”
- Short development cycle
- Testing is understanding

TDD benefits

- Less Debugging (debugging is tedious and expensive)
- Small steps – easier to backtrack (less debugging)
- Small pieces drive developers towards **modularity**
- Bugs caught earlier
- Testing is understanding

TDD vulnerabilities

- Developer writing tests for his own code
- Checking private variables may require dedicated “hacks” that may remain in the code.
- More code
- Longer time to write
- Over reliance on unit testing

TDD tools

- Junit – increases the productivity of writing tests. Supplies mechanisms such as assert to check tests ...
- Mockito – enables programmer to replace none existing classes with minimal effort. Has extra features that increase testing capability – such as checking how the mocked classes where treated.