1

ITSS SOFTWARE DEVELOPMENT/SOFTWARE DESIGN AND CONSTRUCTION

11. COUPLING AND COHESION

Nguyen Thi Thu Trang
trangntt@soict.hust.edu.vn

1

2

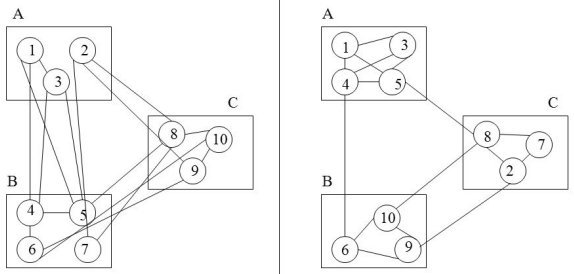
Review: Design Concepts

- Cohesion
  - Degree to which the elements of a module belong together
  - Why are sub-modules placed in the same module? How strongly-related or focused the responsibilities of a single module are?
- Coupling
  - Degree to which each program module relies on each one of the other modules
  - How elements are independent?

2

3

Cohesion within modules and coupling across modules



Bad modularization:
low cohesion, high coupling

Good modularization:
high cohesion, low coupling

3

4

Good design

- Easy for Developing, Testing,
- Easy for Reading, Understanding
- Easy for Communication
- Easy for Extending (add new features)
- Easy for Maintenance

➔ “Loose coupling and tight cohesion”

4

5

How to measure Cohesion and Coupling?

- Design issues
  - Each element can do **lots of things**
  - Interact with **a large number** of other elements
- Measure the interconnectivity or "doing too much"
  - Cohesion: a measure of how much elements do
    - Cohesive military unit: one goal, strong team
  - Coupling: a measure of how elements are connected
    - E.g. Train cars are coupled nicely if they aren't too lose and aren't too tight

➔ **Measured by Cohesion and Coupling Levels**

5

6

Content

1. Coupling Levels
2. Cohesion Levels

6

7

Coupling

- Degree to which components are related
  - On a continuum from *tightly* (worse) to *loosely* (better) coupled
- Tight coupling drawbacks
  - Spaghetti code
  - Assembling components takes longer
  - Components less reusable
  - Increased likelihood that a change to one component forces a change to another

7

8

Coupling Levels/Types

- Content
- Common
- Control
- Stamp
- Data

Tight/High

Loose/Low

Less flexible (worse)

More flexible (better)

8

9

2.1. Content coupling

Content
Common
Control
Stamp
Data

- Using content (data, control) encapsulated within the boundary of another module
  - Directly modifies another's data
  - Refers to local data of another component in terms of numerical displacement
  - Modifies another's code, e.g., jumps into the middle of a routine (goto statement)
- **Worse coupling**

```
public class Database
{
  public SomeType
            theData;
  ...
}
```

```
public class DatabaseCleaner {
  public void cleanupDB(Database
db){
    ...
    db.theData = x;
    ...
  }
```

9

10

Breaking Encapsulation

```
// a well-behaved class:

class Student {
  private String firstName = "Sampath";
}

…

Student stu = new Student();
System.out.println(stu.firstName); // ERROR!
```

But is there a way to make this legal?

10

11

```
import java.lang.reflect.Field;

class Student {
  private String firstName = "Sampath";
}

public class TestReflection {
  public static void main(String args[]) throws Exception {
      Student someStudent = new Student();

      Field fs = someStudent.getClass().getDeclaredField("firstName");
      fs.setAccessible(true);

      System.out.println("Value of " + fs.getName() + ": " +
fs.get(someStudent));

      fs.set(someStudent, "Trisha");
      System.out.println("Value of " + fs.getName()
                         + " after changing: " + fs.get(someStudent));
  }
}
```

```
# output

Value of firstName: Sampath
Value of firstName after changing: Trisha
```

Makes field accessible by both get, se

- *Consequences?*
  So why have this?
- Applications: serialization, processing fxml in JavaFX

11

12

Exercise of Content Coupling

Part of program handles lookup for customer. When customer not found, component adds customer by directly modifying the contents of the data structure containing customer data

=> How to improve?

12

13

Exercise of Content Coupling

Part of program handles lookup for customer.
When customer not found, component adds customer by directly modifying the contents of the data structure containing customer data.

Improvement:
When customer not found, component calls the addCustomer() method that is responsible for maintaining customer data.

13

14

Content Coupling: Issues and Solutions

- Issues
  - Violating the encapsulation/information hiding
  - Nothing is safe
  - Unexpected/Uncontroled processing/results
- Solutions
  - Never modify the control of other modules
  - Access other modules' data through calling functions/sending messages

14

15

Quiz

- Is it content coupling if we provide getter and setter methods for all private attributes?
  - Getters: Only return attributes' values
  - Setters: Set the values of attributes

15

16

2.2. Common coupling

Content
Common
Control
Stamp
Data

- Modules share the same global data area or data structure
- Closely related to content coupling
- **Poor design**

```
public class GlobalData{
   public static SomeType theData;
}

public class DataUserOne{
   public void useTheData(){
      ...
      GlobalData.theData = x;
      ...
   }
}

public class DataUserTwo{
   public void useTheData(){
      ...
      GlobalData.theData = y;
      ...
   }
}
```

16

17

Exercise of Common Coupling

- Process control component maintains current data about state of operation. Gets data from multiple sources. Supplies data to multiple sinks.
- Each source process writes directly to global data store. Each sink process reads directly from global data store.

=> How to improve?

17

18

Exercise of Common Coupling

- Process control component maintains current data about state of operation. Gets data from multiple sources. Supplies data to multiple sinks.
- Each source process writes directly to global data store. Each sink process reads directly from global data store.

=> Improvement:

- Data manager component is responsible for data in data store. Processes send data to and request data from data manager.

18

19

Common Coupling: Issues and Solutions

- Issues
  - Error can spread throughout the system
  - *Reduces readability:* Lack of clear responsibility for data
  - *Reduces maintainability:* Difficult to determine all the components that affect a data element
  - *Bad resuse:* Difficult to reuse components
  - *Not consistent:* Reduces ability to control data accesses
- Solutions
  - Procedural programing: Interact to functions responsible for manage the common data
  - OOP: Let the data belong to a module, and managed by that module

19

20

Content
Common
Control
Stamp
Data

2.3. Control coupling

- Control what another module does by passing arguments (i.e. control parameters)
  - A method does different things depending on the value of a "flag" parameter
- **Bad coupling**

```
public class MyMain{

   public static void myFunc(int mode){
      if (mode == 0) {
         ... do something ...
      }
      else if (mode == 1){
         ... do something else ...
      }
   }

   public static void main(String[]
args){
      myFunc(1);
   }
```

20

21

Control coupling: More examples

```
public class Shape {
    void draw() { assert false; } // stub for draw method
    void draw(int shape, int x1, int y1, int x2, int y2
                         int x3, int y3, int x4, int y4) {
        switch (shape) {
           case 1: ...draw line using (x1, y1), (x2, y2)...
           case 2: ...draw triangle with (x3, y3) as well...
           case 3: ...draw rectangle using all 4...
           case 4: ...draw circle within enclosing rectangle...
        }
    }
}

public class EquilateralTriangle extends Shape {
    private int top_x, top_y, height;
    public void draw() {
        super.draw(2, top_x, top_y, top_x – height/2,
                   top_y – height,
                    top_x + hieght/2, top_y – height,
                    0, 0);
    }
}
```

Passing control information

Note not used

21

22

Control coupling: More examples

- Frequent form: handle all errors
- Seems consistent
- Difficult to use
- Impossible to reuse
- Any change to control: revisit all calls

```
void handleErrors(int errorNumber) {
    String[] messages = {"file not found",
       "folder not found", "file already open",
       "disk error"};
    System.out.print(messages[errorNumber]);
    if (errorNumber == 1 || errorNumber == 2)
        System.exit(3);
    else if (errorNumber == 3)
       closeAllFiles();
    else {
        System.out.println("Cannot recover.");
        System.exit(1);
    }
}
```

22

23

Exercise – Control Coupling

- In your video store, you might eventually create a method like this:
  - **updateCustomer(int action, Customer customer)** where **action** takes on the values **ADD**, **EDIT** or **DELETE**

➔ **Solution?**

23

24

Control Coupling: Issues and Solutions

- Issues
  - *Difficult to understand and maintain*
    - The logic is hard to follow
    - Method becomes very complicated to maintain
    - Gradually extent of param list when new logics defined
  - *Little encapsulation:* Caller must know how the branching works (logic flow)
  - *Bad names:* No names to tell you what's happening; just need to know extra meanings
  - *Bad reuse:* Cannot reuse components by themselves

24

25

Control Coupling: Issues and Solutions (2)

- Solutions
  - Break into different methods for different cases
  - Think of applying inheritance and polymorphism for different types if application
  - Use design patterns for some special purposes

25

26

Content
Common
Control
Stamp
Data

2.4. Stamp coupling

- Passing complex data structure
  - Containing fields which may or may not be used
- May be necessary due to efficiency
  - Avoid by Insightful designers, not lazy programmers

```
public class Employee{
    private String name;
    private double salary;
    ...
}

public class SalaryProcessor{
    public void processSalary(Employee
e){
        ...do stuff with only e.getSalary()...
    }
}

public class NameProcessor {
    public void processName(Employee e){
        ...do stuff with only e.getName()...
    }
}
```

26

27

Stamp Coupling: More examples

- swap(int[] numbers, int pos1, int pos2);
- Attributes of Class:

```
public class Image { … }
public class Buffer { Image a, b; ... }
...
void drawPartB(Buffer full) { ... use just full.b ...}
```

27

28

Stamp Coupling: Broader View

- One module passes non-global **data structure** to another module
- A change in data structure will affect all modules that use it
  - The two module must have some knowledge of the internal workings of other modules that use the same data structure
  - Two modules are stamp coupled if they communicate via a composite data item
    - E.g. record in PASCAL, structure in C, object in Java

28

29

Example of Stamp Coupling

Customer billing system

The print routine of the customer billing accepts a customer data structure as an argument, parses it, and prints the name, address, and billing information.

=> Inprovement?

29

30

Example of Stamp Coupling

Customer Billing System

The print routine of the customer billing accepts a customer data structure as an argument, parses it, and prints the name, address, and billing information.

=> Improvement

The print routine takes the customer name, address, and billing information as an argument.

30

31

Stamp Coupling: Issues and Solutions

- Issues
  - Requires second component to know how to manipulate the data structure (e.g., needs to know about implementation)
  - Piece not as reusable as it could be
  - Piece could have undesired side-effects; could even result in security breaches
- Solutions
  - Only pass elementary required params

31

32

Content
Common
Control
Stamp
Data

2.5. Data coupling

- Each datum of parameter is an elementary piece, and only data shared
- Every argument is simple argument
- **Best coupling (loose)**

```
public class Employee{
  private String name;
  private double salary;
  ...
}

public class SalaryProcessor{
  public void processSalary(double
salary){
    ...do stuff with a copy of e.salary...
  }
}

public class NameProcessor {
  public void processName(String name){
    ...do stuff with a copy of e.name ...
  }
}
```

32

33

Data Coupling: More information

- Modules are independent of each other
- Two modules has no need to know what goes on inside each module
  - Higher level module needs only to know what data to pass its subordinates
  - Subordinate module only need to know what data it requires and what data is return
- Only communicate through passing parameters
  - Non-global variables
  - An elementary data item or a data structure in which all elements are used

33

34

Law of Demeter

Karl Lieberherr ⓘ and colleagues

- Law of Demeter: An object should know as little as possible about the internal structure of other objects with which it interacts – a question of coupling
- Or… "only talk to your immediate friends"
- Closely related to representation exposure and (im)mutability
- Bad example – too-tight chain of coupling between classes

```
general.getColonel().getMajor(m).getCaptain(cap)
    .getSergeant(ser).getPrivate(name).digFoxHole();
```

- Better example

```
general.superviseFoxHole(m, cap, ser, name);
```

34

35

An object should only send messages to …

(More Demeter)

- itself (`this`)
- its instance variables
- its method's parameters
- any object it creates
- any object returned by a call to one of `this`'s methods
- any objects in a collection of the above

Guidelines: not strict rules! But thinking about them will generally help you produce better designs

- notably absent: objects returned by messages sent to other objects

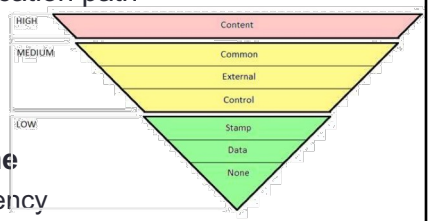
Don't talk to strangers!

35

36

Data Coupling: Advantages

- Easy to write contracts for this and modify module independently
- Simple, easy-to-understand argument lists
- Simple communication path

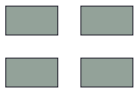


- **Goal: green zone**
- None: no dependency

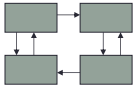
36

37

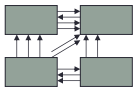
Coupling: Degree of dependence among components



No dependencies



Loosely coupled-some dependencies



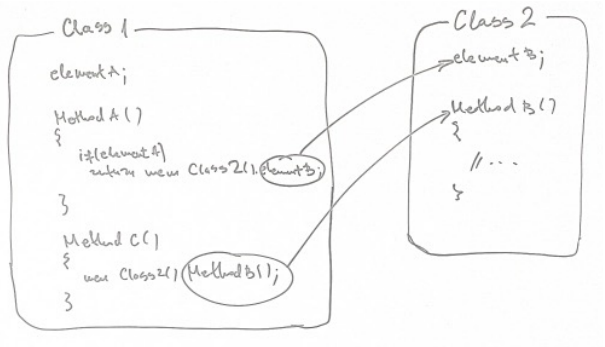
Highly coupled-many dependencies

High coupling makes modifying parts of the system difficult, e.g., modifying a component affects all the components to which the component is connected.

37

38

Quiz: Is it loose or tight coupling?



38

39

Loose coupling is the path to the dark side

- Coupling leads to complexity
- Complexity leads to confusion
- Confusion leads to suffering

- Once you start down the dark path, forever will it dominate your destiny, consume you it will

39

40

Practice: Coupling Levels

- Checking the coupling levels of the existing codebase with explanations

40

41

Content

1. Coupling Levels
2. Cohesion Levels

41

42

God classes

- *God class*: a class that hoards too much of the data or functionality of a system
  - Poor cohesion – little thought about why all of the elements are placed together
  - Only reduces coupling by collapsing multiple modules into one (and thus reducing the dependences between the modules to dependences within a module)
- A god class is an example of an *anti-pattern* – it is a known bad way of doing things

42

43

2. Cohesion

- Definition: The degree to which all elements of a module are directed towards **a single task** and **all elements** directed **towards that task** are contained in a single module.
- Internal glue with which module is constructed
- All elements of component are directed toward and essential for performing the same task
- **High is good**

43

44

Measure cohesion levels on

- Procedural Programming
  - Procedure, **Function**
- Object-Oriented Programming: Any portion of the system
  - Method: Statements, Block of code
  - **Class: Attributes, Methods**
  - Subsystem: Class, Interface

44

45

Cohesion Levels/Types

- Functional
- Communicational
- Procedural
- Temporal
- Logical
- Coincidental

High (Tight)

More cohesive (better)

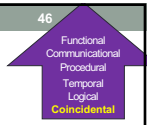
Low (High)

Less cohesive (worse)

45

46

2.1. Coincidental Cohesion



- Elements of the component are **totally unrelated**
  - Only related by their location in source code (**accidental**)
  - One of which is selected by calling module
- Worst cohesion

```
public class MyMain {

   public static void main(String[] args) {
      ... logic for arg processing ...
      ... logic for opening input streams and reading data ...
      ... logic for processing data ...
      ... logic for closing streams and cleanup ...
   }
}
```

46

47

Coincidental cohesion: More Examples

- Just do some random things in a method
  - Print next line
  - Reverse string of characters in second argument
  - Add 7 to 5th argument
  - Convert 4th argument to float
- Putting all of your GUI code in a single form class
  - This was easy to do in Java Swing
  - The bee simulator sample code
- StudentFinancialsWithCoursesAndResidence
  - How about “StudentData”?
  - All of the clauses, especially “and” show this has too many responsibilities

47

48

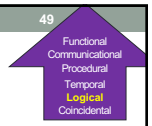
Coincidental cohesion: Issues and Solutions

- Design issues
  - Elements needed to achieve some functionality are scattered throughout the system
  - Hard to maintain, to reuse
- Solutions
  - Move unrelated parts to corresponsibly responsible modules
  - Break unrelated parts into separate modules, each performing one task
  - Use design patterns for some special purposes

48

49

2.2. Logical cohesion



- Parts grouped because they fit into a **logical category** (e.g. they do the same type of thing), but not a functional one
  - They perform similar, but independent computations
  - One of which is selected by the client component
- **Bad cohesion**

```
public class InputHandler{
    public void handleDiskInput(){...}
    public void handleKeyboardInput(){...}
    public void handleControllerInput(){...}
    public void handleMouseInput(){...}
}
public class EventLogger {
    public void logDatabaseError() {...}
    public void logFileReadingError() { ... }
    public void logUserInputError() { ... }
}
```

49

50

Logical Cohesion: More Examples

- Handle/generate all regardless of type
  - handleAllErrors(int errno, const char *msg)
  - sendOutput(int dest, int type, String msg, int inum, float fnum);
    - dest: 0 means stdout, 1 means tape, 2 means printer
    - type: 0 means print msg and inum, 1 means print msg and fnum, 2 means msg only, 3 means inum only, 4 means fnum only
- *Note*
  - *Tempting to engineers: "handle all of the logicalled related stuffs in one place," etc.*
  - *Lead to control coupling*

50

51

Example of Logical Cohesion

- A component reads inputs from tape, disk, and network. All the code for these functions are in the same component.
- Operations are related, but the functions are significantly different.

- => How to improve?

51

52

Example of Logical Cohesion

- A component reads inputs from tape, disk, and network. All the code for these functions are in the same component. Operations are related, but the functions are significantly different.

=> Improvement and charging

- A device component has a read operation that is overridden by sub-class components. The tape sub-class reads from tape. The disk sub-class reads from disk. The network sub-class reads from the network.

52

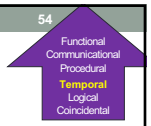
53

Logical cohesion: Issues and Solutions

- Design issues
  - Interface hard to understand
  - Code for more than one actions may be intertwined
  - Intricate code due to module activing differently based on different parameters
  - Hard/impossible to maintain/reuse one piece
- Solutions
  - Break parts into separate modules, each performing one task
  - Use Inheritance, Polymorphism
  - Use Design patterns

53

54



2.3. Temporal cohesion

- Parts grouped because of when they're executed/used (related by timing)
- Not good cohesion

```
public class Startup {
   public void processArgs() {...}
   public void readConfig() {...}
   public void initialize() {...}
   public void openFiles() {...}
   public void displayGreeting() {...}
}

public class Shutdown {
   public void saveConfig() {...}
   public void closeFiles() {...}
   public void cleanup() {...}
}
```

54

55

Temporal Cohesion: More examples

- Classic example: initializing all data structures/state variables
  - `public class Setup`
  - Issue: separating initialization from use, so must maintain two places
- In methods: Document.new
  - Prompt user to save current, clear document, reinitialize state, update recent document list, update title bar
  - All this needs to be done, but with different (reusable) methods, probably in different classes
  - Note: issue is what's done *directly* vs. what's done in a method that's *called* by a more policy-oriented one

55

56

Example of Temporal Cohesion

- A system initialization routine: this routine contains all of the code for initializing all of the parts of the system. Lots of different activities occur, all at init time.

- => How to improve?

56

57

Example of Temporal Cohesion

- A system initialization routine: this routine contains all of the code for initializing all of the parts of the system. Lots of different activities occur, all at init time.

=> How to improvement?

- A system initialization routine sends an initialization message to each component.
- Each component initializes itself at component instantiation time.

57

58

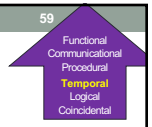
Temporal Cohesion: Issues and Solutions

- Issues
  - Actions weakly related to one another, but strongly related to actions in other modules
  - Hard to maintain and reuse, increases chances of regression fault
    - Code spread out
    - Have to look at numerous modules when a change in a data structure
- Solutions
  - Move parts to corresponding methods/modules
  - Use design patterns for some special purposes

58

59

2.4. Procedural Cohesion



- Parts grouped because they're executed in sequence (their procedure/order of execution)
- Not good cohesion
- Examples
  - ReadPartNumberFromDatabaseAndUpdateRepairRecord
  - FilterChain

```
public class FilterChain {
   public Object filter1(Object o) {...}
   public Object filter2(Object o) {...}
   public Object filter3(Object o) {...}
   public Object filter4(Object o) {...}
}
```

59

60

Procedural Cohesion: More Examples

...
Read part number from database
Update repair record on maintenance file.
...

- May be useful to abstract the intent of this sequence. Make the database and repair record components handle reading and updating. Make component that handles more abstract operation.

60

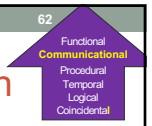
61

Procedural Cohesion: Issues and Solutions

- Issues
  - Actions are still weakly related to one another
  - Doing the operations directly in the method; having a method which calls others to do the real work is far less of a problem
    - Often: no data being passed
  - Low reusability
- Solutions
  - Move parts to corresponding methods/modules
  - Use design patterns for some special purposes

61

62



2.5. Communicational Cohesion

- Parts grouped because they are related to each other by the data that they use
- Acts on the same input data or is concerned with the same output data
- Still low reusability → Break it up

```
public class CoordinateManager {
   calculate new coordinates
   send them to window
}
```

62

63

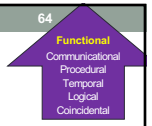
Communicational Cohesion: More Example

- Update record in database and send it to the printer.
  - database.update (record).
  - record.print().

```
public class RecordManager {
   update record in db
   write it to audit trail
}
```

63

64



2.6. Functional Cohesion

- Parts grouped because they work together to perform a well-defined responsibility
  - Every essential element to a single computation is contained in the module
- The name of module will indicate its responsibility
- Best Cohesion (Highest) → Ideal situation

```
public class Stack {
   private Object[] theStack;
   public void push(Object o) {...}
   public Object top() {...}
   public void pop() {...}
}
```

64

65

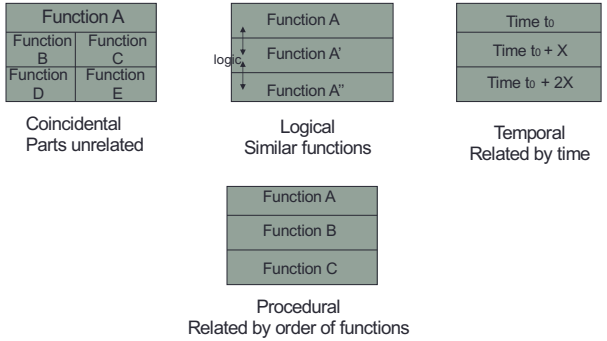
Advantages of Functional Cohesion

- More reusable
- Corrective maintenance easier
  - Fault isolation
  - Reduced regression faults
- Easier to maintain
- Easier to extend product
- E.g.
  - `Furnace.getTemperature()`
    `Electron.computeOrbital()`
  - `SalesCommission.calculate()`

65

66

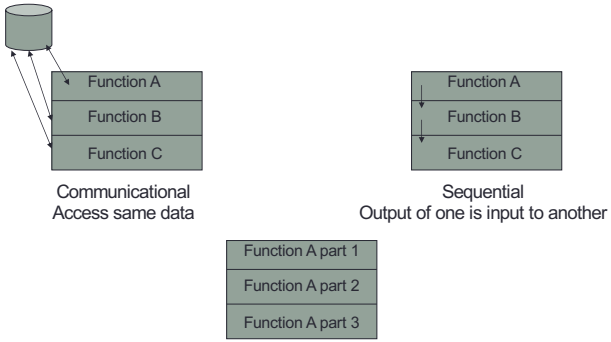
Examples of Cohesion



66

67

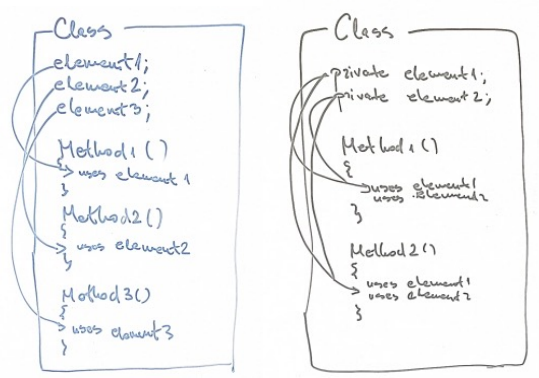
Examples of Cohesion-2



67

68

Which is better?



68

69

Exercise: Cohesion Levels

- Checking the cohesion levels of the existing codebase with explanations

69