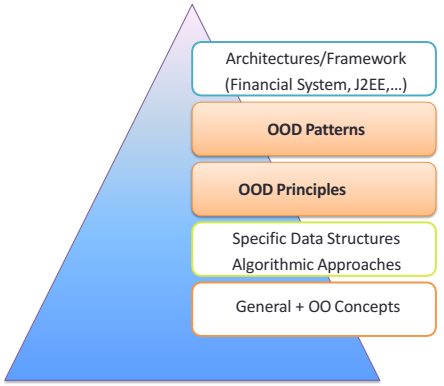
1

IT4490 - SOFTWARE DESIGN AND CONSTRUCTION

10. DESIGN PRINCIPLES

3

Design levels



4

How do you design?

- What principles guide you when you create a design?
- What considerations are important?
- When have you done enough design and can begin implementation?

•Take a piece of paper and write down two principles that guide you - considerations that are important or indicators that you have a good design.

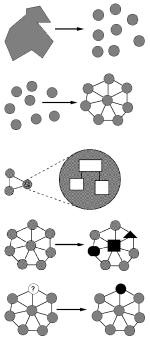
5

Modules

- A *module* is a relatively general term for a class or a type or any kind of design unit in software
- A *modular design* focuses on what modules are defined, what their specifications are, how they relate to each other, but not usually on the implementation of the modules themselves
- Overall, you've been given the modular design so far – and now you have to learn more about how to do the design

Ideals of modular software

- Decomposable – can be broken down into modules to reduce complexity and allow teamwork
- Composable – "Having divided to conquer, we must reunite to rule [M. Jackson]."
- Understandable – one module can be examined, reasoned about, developed, etc. in isolation
- Continuity – a small change in the requirements should affect a small number of modules
- Isolation – an error in one module should be as contained as possible



Modularity

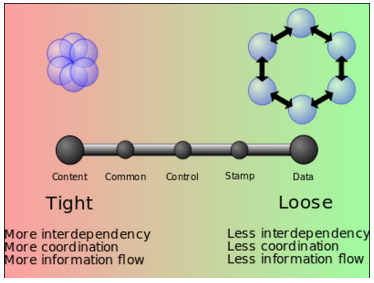
- The goal of design is to partition the system into modules and assign responsibility among the components in a way that:
  - High cohesion within modules, and
  - Loose coupling between modules
- Modularity reduces the total complexity a programmer has to deal with at any one time assuming:
  - Functions are assigned to modules in away that groups similar functions together (Separation of Concerns), and
  - There are small, simple, well-defined interfaces between modules (information hiding)
- The principles of cohesion and coupling are probably the most important design principles for evaluating the effectiveness of a design.

Coupling

- Coupling is the measure of dependency between modules. A dependency exists between two modules if a change in one could require a change in the other.
- The degree of coupling between modules is determined by:
  - The number of interfaces between modules (quantity), and
  - Complexity of each interface (determined by the type of communication) (quality)

Types of Coupling

- Content coupling (also known as Pathological coupling)
- Common coupling
- Control coupling
- Stamp coupling
- Data coupling



Content Coupling

- One module directly references the contents of another
  1. One module modifies the local data or instructions of another
  2. One module refers to local data in another
  3. One branches to a local label of another

Common Coupling

- Two or more modules connected via global data.
  1. One module writes/updates global data that another module reads

Control Coupling

- One module determines the control flow path of another. Example:

```
print(milesTraveled, displayMetricValues)
. . .
public void print(int miles, bool displayMetric) {
    if (displayMetric) {
        System.out.println(. . .);
        . . .
    else { . . .}
}
```

Stamp Coupling (Data-structured coupling)

- Passing a composite data structure to a module that uses only part of it. Example: passing a record with three fields to a module that only needs the first two fields.

Data Coupling

- Modules that share data through parameters.

Cohesion

- Cohesion is a measure of how strongly related the functions or responsibilities of a module are.
- A module has high cohesion if all of its elements are working towards the same goal.

16

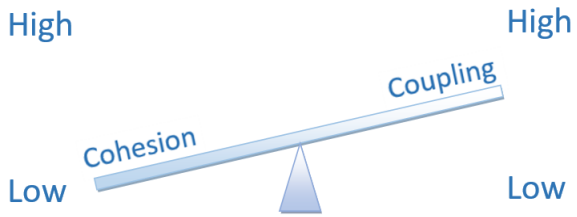
High cohesion, low coupling guideline

- In essence, high cohesion means **keeping parts of a code base that are related to each other in a single place**.
- Low coupling, at the same time, is about **separating unrelated parts of the code base as much as possible**

Benefits of high cohesion and low coupling

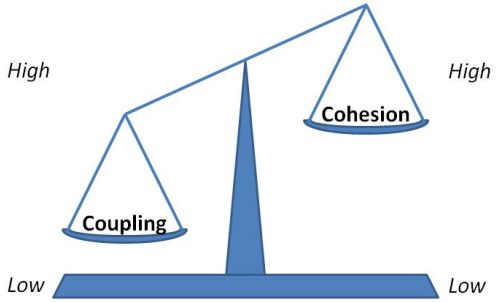
- Modules are easier to read and understand.
- Modules are easier to modify.
- There is an increased potential for reuse
- Modules are easier to develop and test.

Coupling and Cohesion Tend to be Inversely Correlated



When Coupling is high, cohesion tends to be low and vise versa.

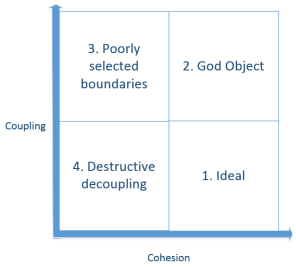
Relationship between Coupling and Cohesion



21

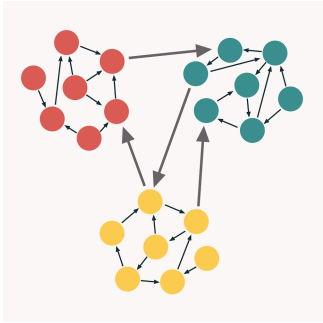
High cohesion, low coupling guideline

- In essence, high cohesion means **keeping parts of a code base that are related to each other in a single place**.
- Low coupling, at the same time, is about **separating unrelated parts of the code base as much as possible**



22

Ideal



23

God Object

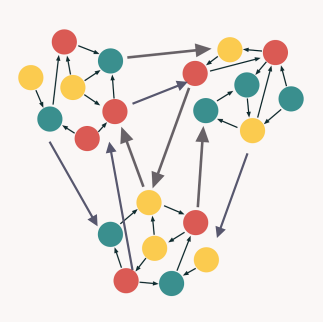
24

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

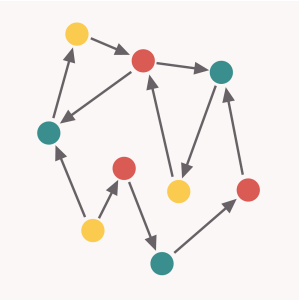
25

Poorly selected boundary



26

Destructive decoupling



27

Example of destructive decoupling

```
public class Order
{
    public Order(IOrderLineFactory factory, IOrderPriceCalculator calculator)
    {
        _factory = factory;
        _calculator = calculator;
    }

    public decimal Amount
    {
        get { return _calculator.CalculateAmount(_lines); }
    }

    public void AddLine(IProduct product, decimal price)
    {
        _lines.Add(_factory.CreateOrderLine(product, price));
    }
}
```

28

Improvement

```
public class Order
{
    public decimal Amount
    {
        get { return _lines.Sum(x => x.Price); }
    }

    public void AddLine(Product product, decimal amount)
    {
        _lines.Add(new OrderLine(product, amount));
    }
}
```

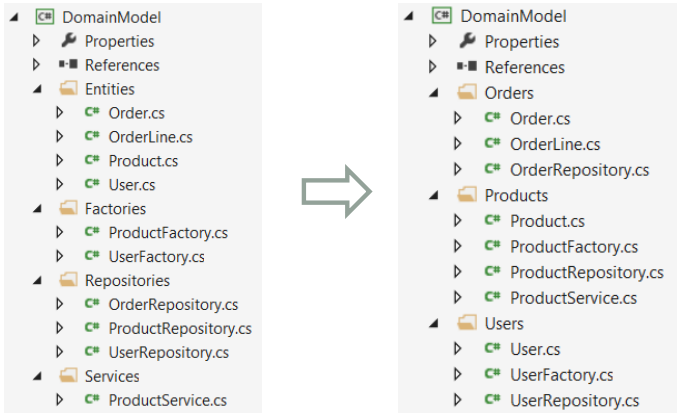
29

Relation between cohesion and coupling

It's impossible to completely decouple a code base without damaging its coherence

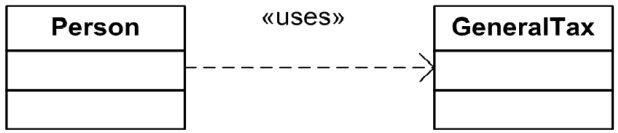
30

Cohesion and coupling on different levels



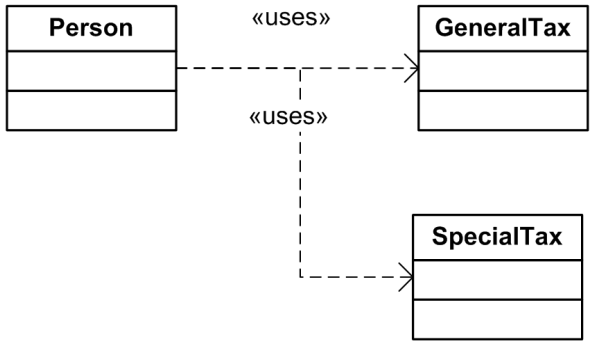
31

Loose coupling example (1/3)



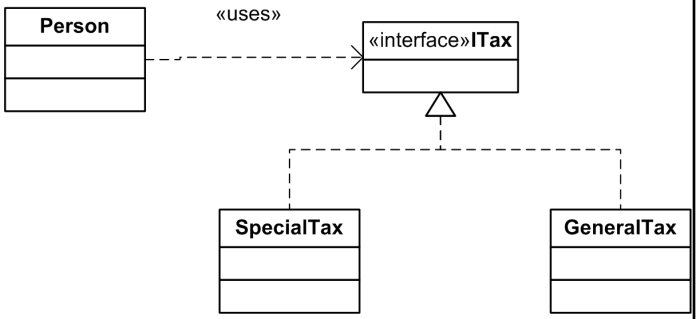
32

Loose coupling example (2/3)



33

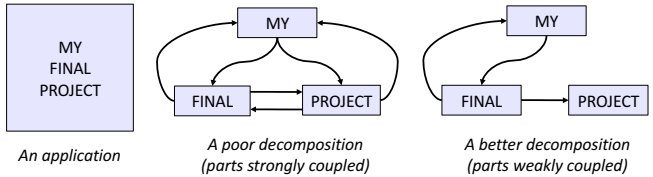
Loose coupling example (3/3)



Coupling

Roughly, the more coupled k modules are, the more one needs to think of them as a single, larger module

- How are modules dependent on one another?
  - Statically (in the code)? Dynamically (at run-time)? And more
  - Ideally, split design into parts that don't interact much



An application | *A poor decomposition (parts strongly coupled)* | *A better decomposition (parts weakly coupled)*

- An artist's rendition – to really assess coupling one needs to know what the arrows are, etc.

36

Different kinds of dependences

- Aggregation – "is part of" is a field that is a sub-part
  - Ex: A car has an engine
- Composition – "is entirely made of" has the parts live and die with the whole
  - Ex: A book has pages (but perhaps the book cannot exist without the pages, and the pages cannot exist without the book)
- Subtyping – "is-a" is for substitutability
- Invokes – "executes" is for having a computation performed
- In other words, there are lots of different kinds of arrows (dependences) and clarifying them is crucial

37

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);
```

38

Law of Demeter

A method "M" of an object "O" should invoke *only* the the methods of the following kinds of objects:

- itself
- its parameters
- any objects it creates/instantiates
- its direct component objects

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

39

Without Law of Demeter?

objectA.getObjectB().getObjectC().doSomething();

- In the future, the class ObjectA may no longer need to carry a reference to ObjectB.
- In the future, the class ObjectB may no longer need to carry a reference to ObjectC.
- The doSomething() method in the ObjectC class may go away, or change.
- If the intent of your class is to be reusable, you can never reuse your class without also requiring ObjectA, ObjectB, and ObjectC to be shipped with your class.

40

Law of Demeter – benefits

- Your classes will be "loosely coupled"; your dependencies are reduced.
- Reusing your classes will be easier.
- Your classes are less subject to changes in other classes.
- Your code will be easier to test.
- Classes designed this way have been proven to have fewer errors.

41

Example of LoD

```
public class Customer {
  private String firstName;
  private String lastName;
  private Wallet myWallet;

  public String getFirstName(){
        return firstName;
  }
  public String getLastName(){
        return lastName;
  }
  public Wallet getWallet(){
        return myWallet;
  }
}
```

42

Example of LoD

```
public class Wallet {
  private float value;
  public float getTotalMoney() {
        return value;
  }
  public void setTotalMoney(float newValue) {
        value = newValue;
  }
  public void addMoney(float deposit) {
        value += deposit;
  }
  public void subtractMoney(float debit) {
        value -= debit;
  }
}
```

43

Example of LoD

```
// code from some method inside the Paperboy class...
payment = 2.00; // “I want my two dollars!”
Wallet theWallet = myCustomer.getWallet();
if (theWallet.getTotalMoney() > payment) {
        theWallet.subtractMoney(payment);
} else {
        // come back later and get my money
}
```

Is this Bad? Why?

44

Example of LoD – Improvement

```
public class Customer {
  private String firstName;
  private String lastName;
  private Wallet myWallet;
  public String getFirstName(){
        return firstName;
  }
  public String getLastName(){
        return lastName;
  }
  public float getPayment(float bill) {
        if (myWallet != null) {
              if (myWallet.getTotalMoney() > bill) {
                    theWallet.subtractMoney(payment);
                    return payment;
              }
        }
  }
}
```

45

Example of LoD – Improvement

```
// code from some method inside the Paperboy class...
payment = 2.00; // “I want my two dollars!”
paidAmount = myCustomer.getPayment(payment);
if (paidAmount == payment) {
        // say thank you and give customer a receipt
} else {
        // come back later and get my money
}
```

Why Is This Better?

46

Another example of LoD

```
public class Band {
   private Singer singer;
   private Drummer drummer;
   private Guitarist guitarist;
}
```

47

Another example of LoD

```
class TourPromoter {
  public String makePosterText(Band band) {
    String guitaristsName =  band.getGuitarist().getName();
    String drummersName = band.getDrummer().getName();
    String singersName = band.getSinger().getName();
    StringBuilder posterText = new StringBuilder();

    posterText.append(band.getName()
    posterText.append(" featuring: ");
    posterText.append(guitaristsName);
    posterText.append(", ");
    posterText.append(singersName);
    posterText.append(", ")
    posterText.append(drummersName);
    posterText.append(", ")
    posterText.append("Tickets £50.");

    return posterText.toString();
  }
}
```

48

Another example of LoD – Improvement

```
public class Band {
    private Singer singer;
    private Drummer drummer;
    private Guitarist guitarist;

    public String[] getMembers() {
        return {
                    singer.getName(),
                    drummer.getName(),
                    guitarist.getName()};
                }
}
```

49

Another example of LoD – Improvement

```
public class TourPromoter {
    public String makePosterText(Band band) {
        StringBuilder posterText = new StringBuilder();

        posterText.append(band.getName());
        posterText.append(" featuring: ");
        for(String member: band.getMembers()) {
            posterText.append(member);
            posterText.append(", ");
        }
        posterText.append("Tickets: £50");

        return posterText.toString();
    }
}
```

50

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

51

Design exercise

- Write a typing break reminder program
  - Offer the hard-working user occasional reminders of the health issues, and encourage the user to take a break from typing
- Naive design
  - Make a method to display messages and offer exercises
  - Make a loop to call that method from time to time

  (Let's ignore multi-threaded solutions for this discussion)

52

TimeToStretch suggests exercises

```
public class TimeToStretch {
  public void run() {
    System.out.println("Stop typing!");
    suggestExercise();
  }
  public void suggestExercise() {
  ...
  }
}
```

53

Timer calls run() periodically

```
public class Timer {
  private TimeToStretch tts = new TimeToStretch();
  public void start() {
    while (true) {
      ...
      if (enoughTimeHasPassed) {
        tts.run();
      }
      ...
    }
  }
}
```

54

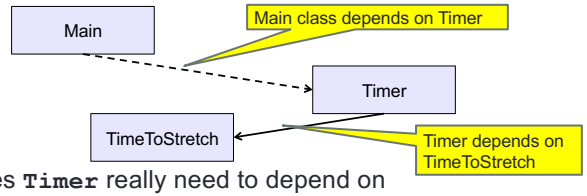
Main class puts it together

```
class Main {
  public static void main(String[] args) {
    Timer t = new Timer();
    t.start();
  }
}
```

55

Module dependency diagram

- An arrow in a module dependency diagram indicates "depends on" or "knows about" – simplistically, "any name mentioned in the source code"



- Does `Timer` really need to depend on `TimeToStretch`?
- Is Timer re-usable in a new context?

56

Decoupling

- `Timer` needs to call the `run` method
  - `Timer` doesn't need to know what the `run` method does
- Weaken the dependency of `Timer` on `TimeToStretch`
- Introduce a weaker specification, in the form of an interface or abstract class

```
public abstract class TimerTask {
    public abstract void run();
}
```

- `Timer` only needs to know that something (e.g., `TimeToStretch`) meets the `TimerTask` specification

57

TimeToStretch (version 2)

```
public class TimeToStretch extends TimerTask {
  public void run() {
    System.out.println("Stop typing!");
    suggestExercise();
  }

  public void suggestExercise() {
    ...
  }
}
```

58

Timer v2

```
public class Timer {
  private TimerTask task;
  public Timer(TimerTask task) { this.task = task; }
  public void setTask(TimerTask task){this.task = task;}
  public void start() {
    while (true) {
      ...
      if (enoughTime)
            task.run();
    }
  }
}
```

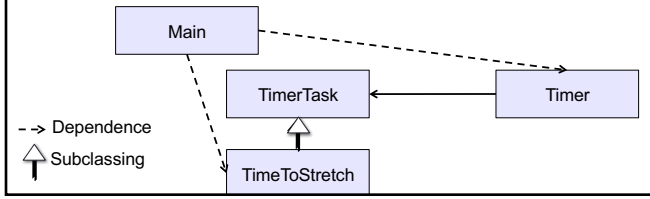
- Main creates the `TimeToStretch` object and passes it to `Timer`

```
Timer t = new Timer(new TimeToStretch());
t.start();
t.setTask(new TimeToSave());
t.start();
```

59

Module dependency diagram

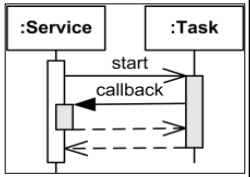
- `Main` still depends on `Timer` (is this necessary?)
- `Main` depends on the constructor for `TimeToStretch`
- `Timer` depends on `TimerTask`, not `TimeToStretch`
  - Unaffected by implementation details of `TimeToStretch`
  - Now `Timer` is much easier to reuse



60

callbacks

- **`TimeToStretch`** creates a **`Timer`**, and passes in a reference to itself so the **`Timer`** can call it back
- This is a *callback* – a method call from a module to a client that notifies about some condition
- Use a callback to invert a dependency
  - Inverted dependency: **`TimeToStretch`** depends on **`Timer`** (not vice versa)
  - Side benefit: **`Main`** does not depend on **`Timer`**



A synchronous callback.
Time increases downward.
Solid lines: calls
Dotted lines: returns

62

TimeToStretch v3

```
public class TimeToStretch extends TimerTask {
    private Timer timer;
    public TimeToStretch() {
        timer = new Timer(this);
    }
    public void start() {
        timer.start();
    }
    public void run() {
        System.out.println("Stop typing!");
        suggestExercise();
    }
    ...
}
```

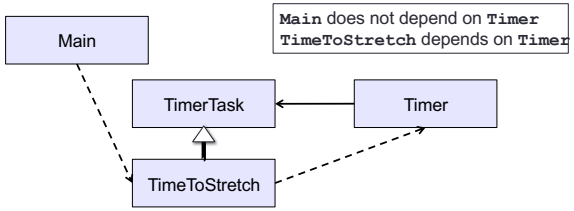
Register interest with the timer

Callback entry point

63

Main v3

- `TimeToStretch tts = new TimeToStretch();`
  `tts.start();`
- Use a callback to invert a dependency
- This diagram shows the inversion of the dependency between **`Timer`** and **`TimeToStretch`** (compared to v1)



64

How do we design classes?

- One common approach to class identification is to consider the specifications
- In particular, it is often the case that
  - *nouns* are potential classes, objects, fields
  - *verbs* are potential methods or responsibilities of a class

65

Design exercise

- Suppose we are writing a birthday-reminder application that tracks a set of people and their birthdays, providing reminders of whose birthdays are on a given day
- What classes are we likely to want to have? Why?

Class shout-out about classes

66

More detail for those classes

- What fields do they have?
- What constructors do they have?
- What methods do they provide?
- What invariants should we guarantee?

In small groups, ~5 minutes