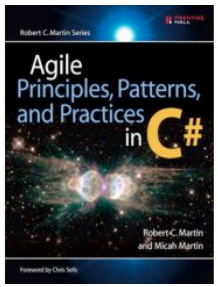
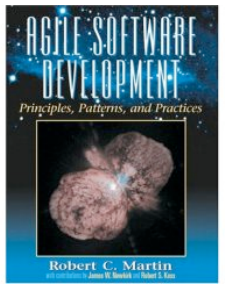
Principles and Rules of Object-Oriented Design

Bibliography





Signs of Rotting Design [Martin, 2002]

- Rigidity
  - code difficult to change (*Continuity*)
  - management reluctance to change anything becomes policy
- Fragility
  - code breaks in unexpected places (*Protection*)
  - even small changes can cause cascading breaks
- Immobility
  - code is so tangled that it's impossible to reuse anything (*Composability*)
    - lots of semantical (or even syntactical) duplication
- Viscosity
  - much easier to hack than to preserve original design
    - "easy to do the wrong thing, but hard to do the right thing" (R.Martin)
  - **software viscosity** and **environment** viscosity

Signs of Rotting Design (2) [Martin, 2002]

- Needles Complexity
  - make design more general than needed
    - e.g. interfaces and/or abstract classes with just one implementor
  - constructs and mechanisms that are never used
- Needles Repetition
  - copy-paste-adapt
    - bugs are cloned as well
  - sign of a missing abstraction
- Opacity
  - convoluted manner of writing code (hampers understandability)
  - *pair-programming* and *refactoring* may help fighting against opaque code

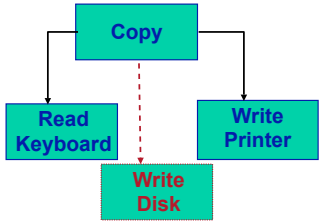
Causes of Rotting Design

1. Changing Requirements
   - is inevitable
   - both better designs and poor designs have to face the changes;
   - good designs are stable

> *All systems change during their life-cycles.*
> *This must be borne in mind when developing systems expected to last longer than the first version.*
> **I. Jacobson**, OOSE, 1992

2. Dependency Management
   - the issue of coupling and cohesion
   - It can be controlled!
     - create *dependency firewalls*

Example of Rigidity and Immobility




```
enum OutputDevice {printer, disk};
void Copy(OutputDevice dev){
   int c;
   while((c = ReadKeyboard())!= EOF)
      if(dev == printer)
         WritePrinter(c);
      else
         WriteDisk(c);
}
```

```
void Copy(){
   int c;
   while ((c = ReadKeyboard()) != EOF)
     WritePrinter(c);
}
```

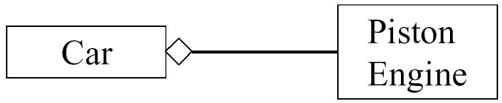

Open-Closed Principle (OCP)

> *Software entities should be open for extension, but closed for modification*
> **B. Meyer**, 1988 / quoted by **R. Martin**, 1996

- Be open for extension
  - module's behavior can be extended
- Be closed for modification
  - source code for the module must not be changes

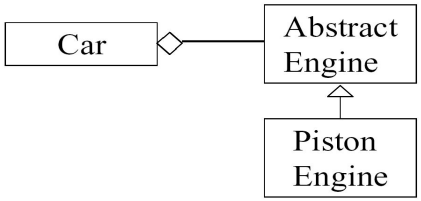
- *Modules should be written so they can be extended without requiring them to be modified*

Open the door ...



- How to make the Car run efficiently with a TurboEngine?
- Only by changing the Car!
  - ...in the given design

... But Keep It Closed!



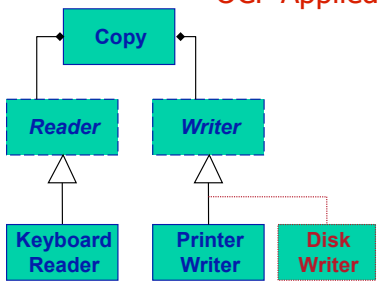
- A class must not depend on a concrete class!
- It must depend on an abstract class ...
- ...using polymorphic dependencies (calls)

OCP Heuristics

RTTI is Ugly and Dangerous!

- RTTI is ugly and dangerous
  - RTTI = **R**un-**T**ime **T**ype **I**nformation
  - If a module tries to dynamically cast a base class pointer to several derived classes, any time you extend the inheritance hierarchy, you need to change the module
  - recognize them by type `switch` or `if-else-if` structures
- Not all these situations violate OCP all the time
  - when used only as a "filter"

OCP Applied on Example




```
class Reader {
  public:
    virtual int read()=0;
};

class Writer {
  public:
    virtual void write(int)=0;
};

void Copy(Reader& r, Writer& w){
   int c;
   while((c = r.read()) != EOF)
     w.write(c);
}
```

What if we decide to change that only each second character is written?

Strategic Closure

No significant program can be 100% closed

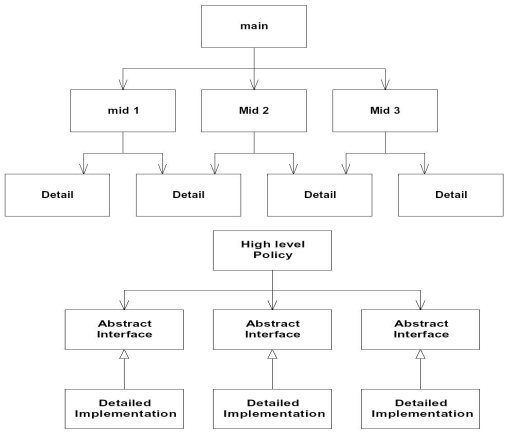
R.Martin

- Closure not *complete* but **strategic**

1. Abstraction to gain explicit closure
   - provide class methods which can be dynamically invoked
     - to determine **general policy decisions**
   - design using abstract ancestor classes
2. Use "Data-Driven" approach to achieve closure
   - place volatile policy decisions in a separate location
     - e.g. a file or a separate object
   - minimizes future change locations

Procedural vs. OO Architecture

Procedural Architecture



Object-Oriented Architecture

Dependency Inversion Principle

I. High-level modules should ***not*** depend on low-level modules.
Both should depend on abstractions.
II. Abstractions should not depend on details.
Details should depend on abstractions

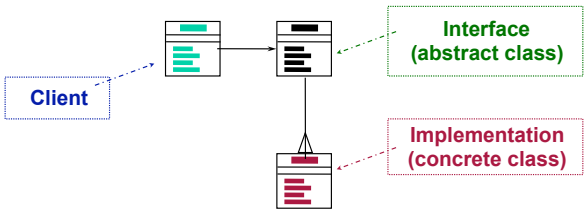
R. Martin, 1996

- A base class in an inheritance hierarchy should not know any of its subclasses
- Modules with detailed implementations are not depended upon, but depend themselves upon abstractions
- OCP states the goal; DIP states the mechanism;

DIP Related Heuristic

Design to an interface, not an implementation!

- Use inheritance to avoid direct bindings to classes:



Design to an Interface

- **Abstract classes/interfaces:**
  - tend to change less frequently
  - abstractions are 'hinge points' where it is easier to extend/modify
  - shouldn't have to modify classes/interfaces that represent the abstraction (OCP)
- Exceptions
  - Some classes are very unlikely to change;
    - therefore little benefit to inserting abstraction layer
    - Example: String class
  - In cases like this can use concrete class directly
    - as in Java or C++

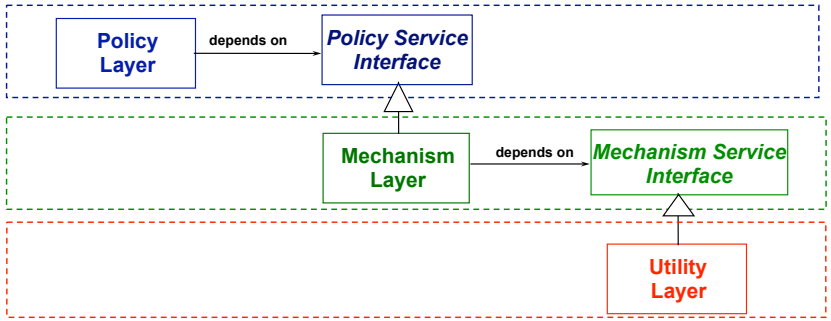
DIP Related Heuristic (2)

Avoid Transitive Dependencies

- Avoid structures in which higher-level layers depend on lower-level abstractions:
  - In example below, Policy layer is ultimately dependent on Utility layer.

Solution to Transitive Dependencies

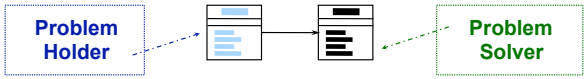
- Use inheritance and abstract ancestor classes to effectively eliminate transitive dependencies:
  - …also a matter of **interface ownership**



DIP - Related Heuristic

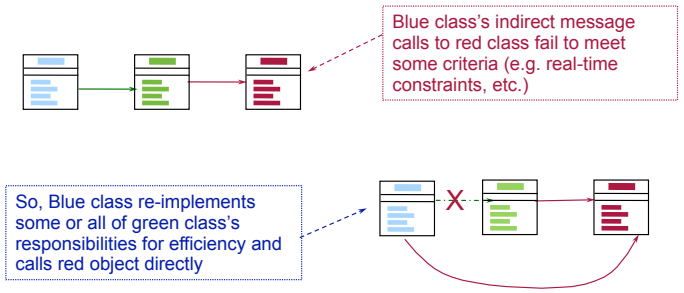
When in doubt, add a level of indirection

- If you cannot find a satisfactory solution for the class you are designing, try delegating responsibility to one or more classes:

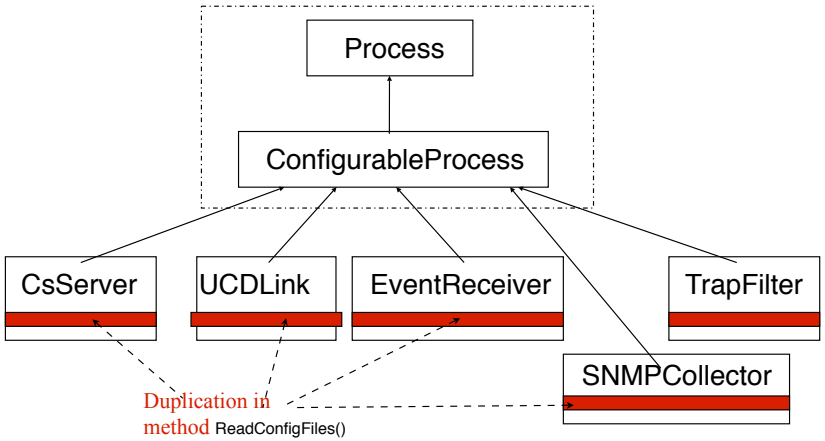


When in doubt ...

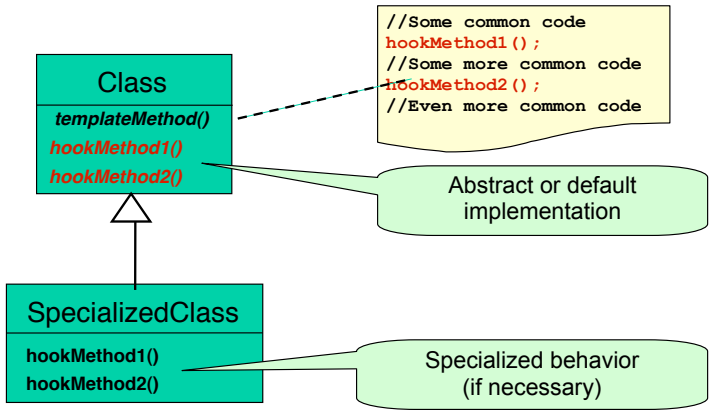
- It is generally easier to remove or by-pass existing levels of indirection than it is to add them later:

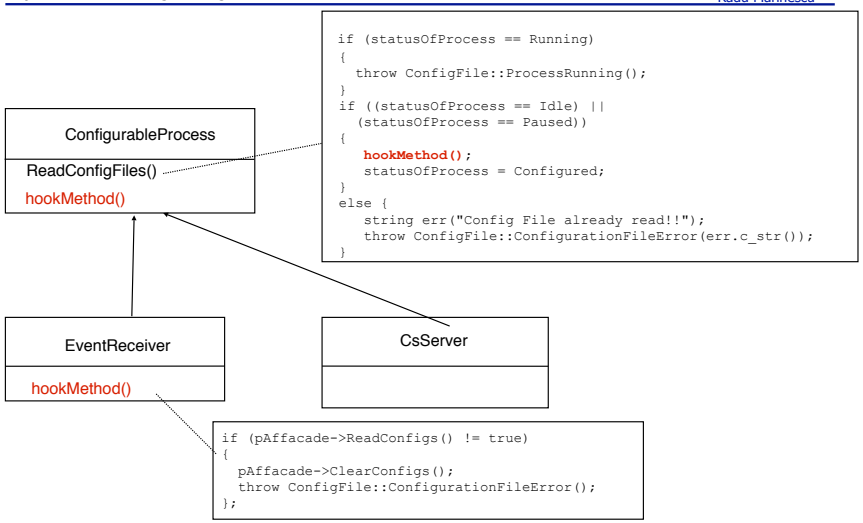


Initial Design with Duplicated Methods

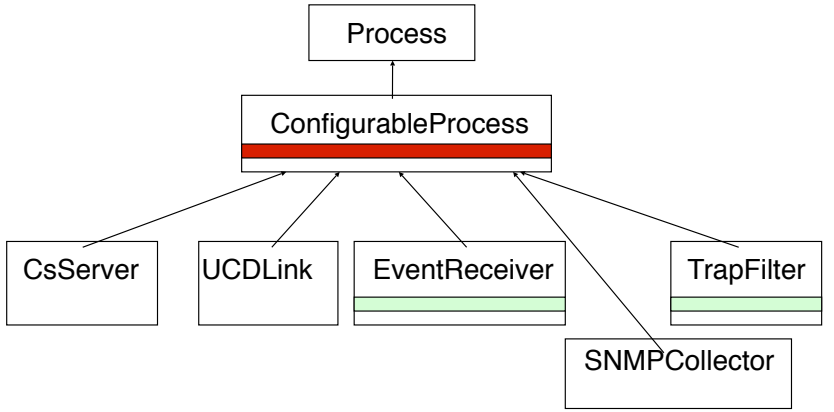


Template Method Pattern





Duplication Resolved



Liskov Substitution Principle (LSP)

- The key of OCP: Abstraction and Polymorphism
  - Implemented by inheritance
  - How do we measure the quality of inheritance?

Inheritance should ensure that any property proved about supertype objects also holds for subtype objects
B. Liskov, 1987

Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.
R. Martin, 1996

Inheritance Appears Simple

```
class Bird {                      // has beak, wings,...
  public: virtual void fly();  // Bird can fly
};

class Parrot : public Bird {      // Parrot is a bird
  public: virtual void mimic();  // Can Repeat words...
};

// ...
Parrot mypet;
mypet.mimic();    // my pet being a parrot can Mimic()
mypet.fly();      // my pet "is-a" bird, can fly
```


Penguins Fail to Fly!

```
class Penguin : public Bird {
   public: void fly() {
     error ("Penguins don't fly!"); }
};
void PlayWithBird (Bird& abird) {
    abird.fly();     // OK if Parrot.
    // if bird happens to be Penguin...OOOPS!!
}
```

- Does **not** model: "*Penguins can't fly*"
- It models "*Penguins may fly, but if they try it is error*"
- Run-time error if attempt to fly → not desirable
  - Think about Substitutability - Fails LSP

Design by Contract

- Advertised Behavior of an object:
  - advertised **Requirements** (Preconditions)
  - advertised **Promises** (Postconditions)

When redefining a method in a derivate class, you may only replace its precondition by a weaker one, *and its* postcondition by a stronger one
B. Meyer, 1988

Derived class services should **require no more** and **promise no less**

```
int Base::f(int x);
// REQUIRE: x is odd
// PROMISE: return even int
```

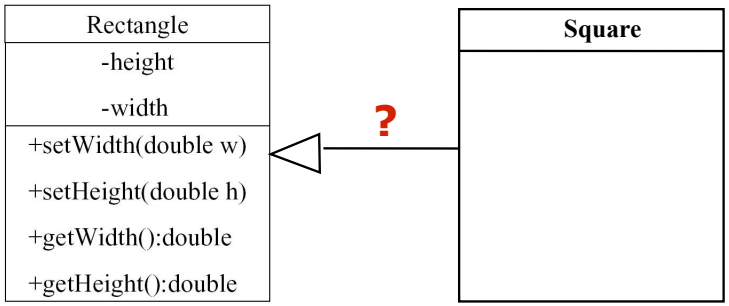
```
int Derived::f(int x);
// REQUIRE: x is int
// PROMISE: return 8
```

Design by Contract

- Impose a certain obligation to be guaranteed on entry by any client module that calls it: the routine's **precondition**
- Guarantee a certain property on exit: the routine's postcondition
- Maintain a certain property, assumed on entry and guaranteed on exit: the class invariant

- What does it **expect**?
- What does it **guarantee**?
- What does it **maintain**?

Square IS-A Rectangle?



- Should I inherit Square from Rectangle?

The Answer is ...

- Override **`setHeight`** and **`setWidth`**
  - duplicated code...
  - static binding (in C++)
    - `void f(Rectangle& r) { r.setHeight(5); }`
    - change base class to set methods `virtual`

- The real problem

```
void g(Rectangle& r) {
  r.setWidth(5); r.setHeight(4);
  // How large is the area?
}
```

  - 20! ... Are you sure? ;-)

IS-A relationship refers to the **BEHAVIOR** of the class!

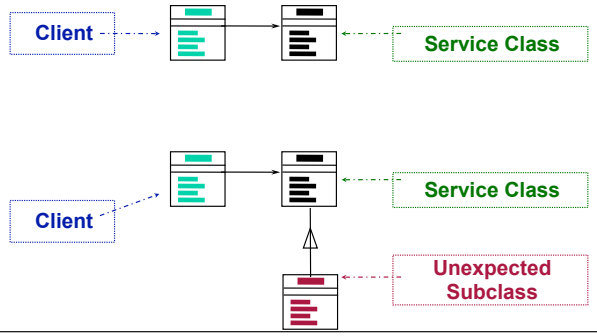
LSP is about Semantics and Replacement

- Understand before you design
  - The meaning and purpose of every method and class must be clearly documented
  - Lack of user understanding will induce de facto violations of LSP

- Replaceability is crucial
  - Whenever any class is referenced by any code in any system, any future or existing subclasses of that class must be 100% replaceable
  - Because, sooner or later, someone **will** substitute a subclass;
    - it's almost inevitable.

LSP and Replaceability

- Any code which can legally call another class's methods
  - must be able to substitute any subclass of that class without modification:



LSP Related Heuristic (2)

It is illegal for a derived class, to override a base-class method with a NOP method

- NOP = a method that does nothing
- Solution 1: Inverse Inheritance Relation
  - if the initial base-class has only additional behavior
    - e.g. `Dog – DogNoWag`
- Solution 2: Extract Common Base-Class
  - if both initial and derived classes have different behaviors
  - for `Penguins` → `Birds`, `FlyingBirds`, `Penguins`
- Classes with bad state
  - e.g. stupid or paralyzed dogs...

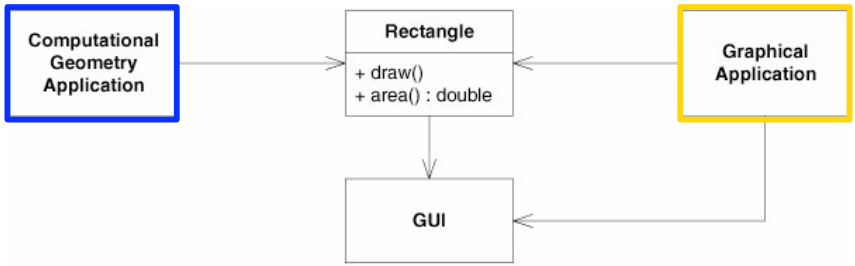
Single Responsibility Principle (SRP)

A class should have only one reason to change!
R. Martin

- A Responsibility is a reasons to change
- Single Responsibility = increased cohesion
- Not following results in needless dependencies
  - More reasons to change.
  - Rigidity, Immobility
- Can be tricky to get granularity right

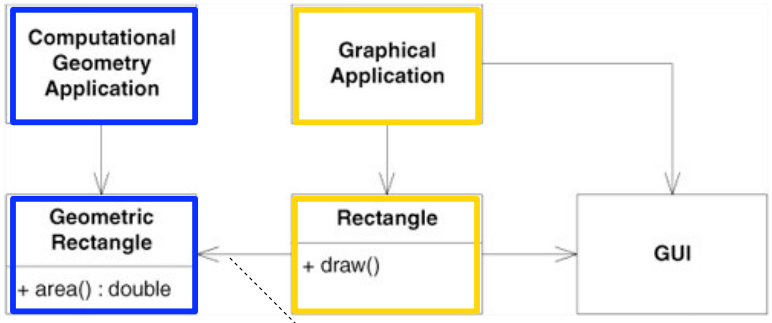
When SRP is violated...

- The Rectangle has 2 responsibilities ... due to the **various clients**



- The **bad part**: ComputationalGeometryApplication depends now on GUI!

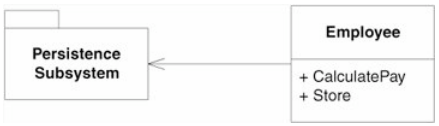
When SRP is considered...



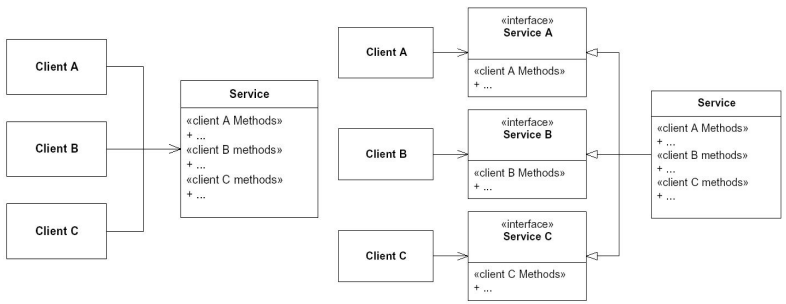
could be either **aggregation** or **inheritance**!

Common Violation of SRP

- mix business rules and persistence control
  - should almost never be mixed
  - business rules change often ; persistency rules change sometimes but always for different reasons



Clients should depend on slim interfaces...



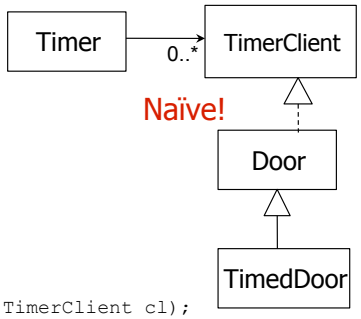
Interface Segregation Principle

Clients should not be forced to depend upon interfaces that they do not use.
R. Martin, 1996

- *Many client-specific interfaces are better than one general purpose interface*
- Consequence:
  - impact of changes to one interface aren't as big if interface is smaller
  - interface pollution

ISP Example

- Door and Timed Door
  - lock() , unlock() , isOpen()
  - TimedDoor beeps when door is open for too long




```
clas Timer {
  public void register(int timeout, TimerClient cl);
  // …
};

interface TimerClient { void OnTimeOut()=0; }
```


What Can Happen?

- What if...

  ....I close the door before the timeout and then open it again ? :))

-> we need to identify the timeout request -> change affects the interface of TimerClient -> **all clients get affected!**

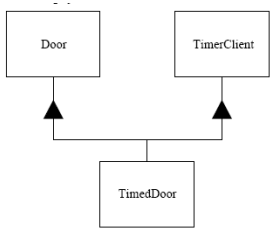
```
public class Timer {
  public void Register(int timeout,
                       int timeOutId,
                       TimerClient client)
  {/*code*/}
}

public interface TimerClient
{
  void TimeOut(int timeOutID);
}
```

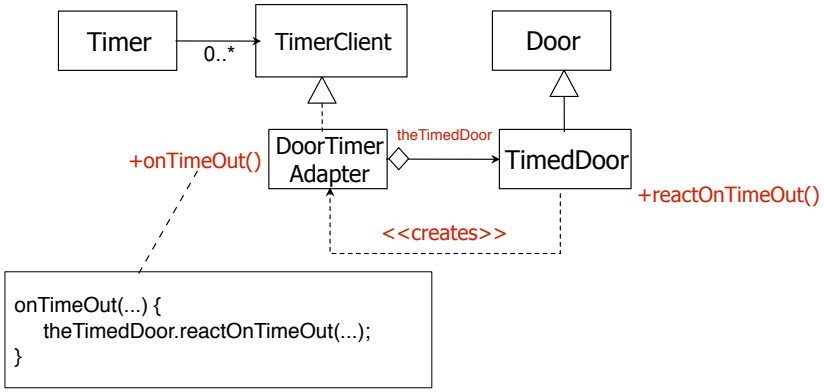
Why should a Door subclass, that is not related to timing be affected by this change?!!

Separation through Multiple Interfaces

- … in fact implementation of multiple interfaces



Separation Through Delegation

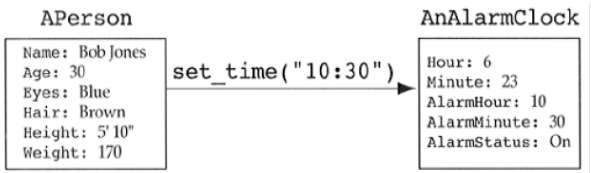


Relaţii intre Clase si Obiecte

Tipuri de Relatii

- Relatia USES
  - Relatia ASSOCIATE
  - Relatia CONTAINS
  - Relatia IS (de mostenire)

Implementare Relatiei USES



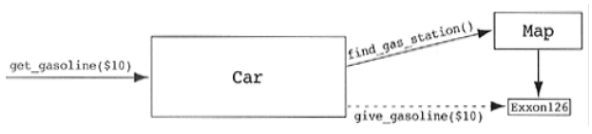
1. Continere
2. Referinta
3. Mapare
4. Parametrii
5. Prin Constructie (Locala)
6. Prin Date Globale

Prin Continere

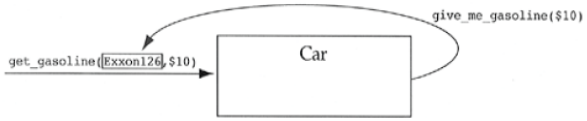
Prin Referinta

- “Oracolul din Delfi” ii spune cine e obietul cu care sa comunice

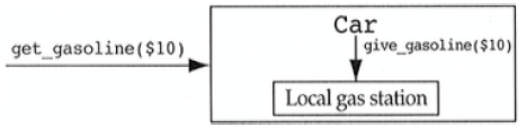
Prin Mapper



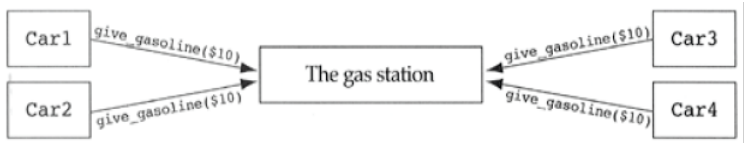
Prin Parametrii



Prin Constructie

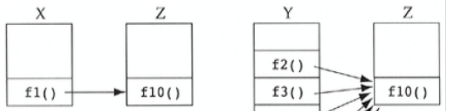


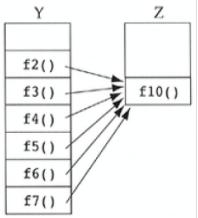
Prin Date Globale



Ce conteaza cand e vorba de cuplaj?

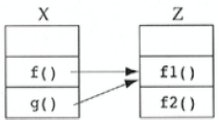
- In care din urmatoarele situatii cuplajul este mai redus?

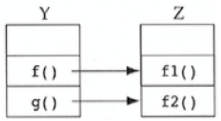




FANOUT = Cate Metode X Cate Apeluri

Nr. Apelanti!





Nr. Apelati!

Euristici Legate De Cuplaj

Reduceti **numarul claselor** de care depinde fiecare clasa!

Reduceti **numarul de metode distincte** apelate dintr-o clasa data!

Reduceti **numarul de apeluri catre alte clase** pentru o clasa data!

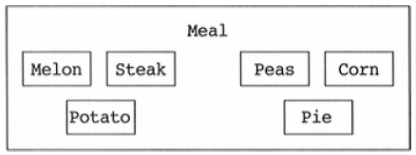
Reduceti **valoare FANOUT** pentru o clasa data!

Relatia de CONTINERE (CONTAINMENT)

Daca o clasa, contine obiecte ale altei clase, atunci clasa care contine obiectele trebuie sa foloseasca acele obiecte. Ea si numai ea!

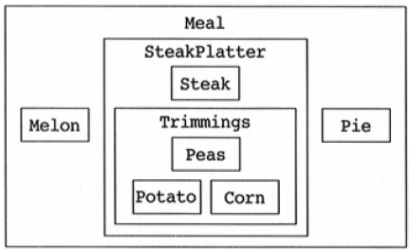
- ... altfel relatia de continere e falsa!
  - fie clasa care contine nu are nevoie de respectivele date
  - fie ele sunt plasate gresit...atunci cand sunt folosite mai mult de catre o alta clasa.

Care clasa ai prefera sa o **utilizezi ?**



A

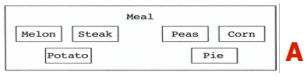
B



Raspunsul Corect:

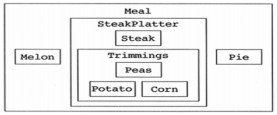
NU CONTEAZA!

Care clasa ai prefera sa o implementezi ?



A

B



Majoritate metodelor unei clase ar trebui sa foloseasca **majoritatea datelor** clasei in **majoritatea timpului**

- In general ar fi bine ca o clasa sa nu contina mai mult de **6 obiecte** ca atribute
- Ce se intampla daca ***cateva*** *metode folosesc majoritatea datelor in majoritatea timpului ?* ;-)

Forma ideala a ierarhiilor de continere

Organizati relatii de continere in ierarhii **adanci** si **inguste**

- ... pentru ca nu mai accesam datele partilor ci identificam servicii ale acestora
- castigam flexibilitatea → modalitati multiple de reutilizare

Constrangeri Semantice

- Ce ne facem daca avem limitari in privinta compunerii felurilor de mancare? ☺
- **Solutia I**
  - nu las sa se construiasca obiectul
  - constrangerea semantica este plasata in constructorul obiectul care agrega
- **Solutia II**
  - constructorul permitea construirea unor obiecte “ilegale”
  - testarea se face la nivelul metodelor care folosesc obiectul

... o sa vedeti voi cand vorbim despre *Composite* ;-)

Shy Code

- Don't reveal yourself to others
  - “**Information Hiding**” modularization rule
- Don't interact with too many people
  - “**Few Interfaces**” modularization rule
- Spy, dissidents and revolutionaries
  - eliminating interactions protects anyone
- The General contractor example
  - he must manage subcontractors

Law of Demeter

- Weak Form

Inside of a method M of a class C, data can be accessed in and messages can be sent to only the following objects:

  - **this** and **super**
  - **data members** of class C
  - **parameters** of the method M
  - **object created** within M
    - by calling directly a constructor
    - by calling a method that creates the object
  - **global variables**

- Strong Form:

In addition to the Weak Form, you are not allowed to access directly inherited members

Demeter’s Law on Example

```
class Demeter {
private:
  A *a;
public:
  // …
  void example(B& b);

void Demeter::example(B& b) {
  C *c;
  c = func();          // itself
  b.invert();          // passed parameters
  a = new A();
  a->setActive();      // created objects
  c->print();          // directly held component objects
}
```

Any methods of an object should call only methods belonging to:

Example of LoD Violation

```
class Course {
     Instructor boring = new Instructor();
     int pay = 5;
     public Instructor getInstructor() { return boring; }
     public Instructor getNewInstructor() {return new Instructor(); }
     public int getPay() {return pay; }
}

class C {
     Course test = new Course();

     public void badM() { test.getInstructor().fired(); }

     public void goodM() { test.getNewInstructor().hired(); }

     public int goodOrBadM?()  { return test.getpay() + 10; }
     }
}
```


How to eliminate the LoD violation?

```
class Course {
     Instructor boring = new Instructor();
     int pay = 5;

     public Instructor fireInstructor() { boring.fired(); }
     public Instructor getNewInstructor() { return new Instructor();}
     public int getPay() { return pay ; }
}

class C {
     Course test = new Course();

     public void reformedBadM() {  test.fireInstructor(); }

     public void goodM() { test.getNewInstructor().hired(); }

     public int goodOrBadM() { return  test.getpay() + 10; }
}
```


The Law of Demeter for Children ;-)

- ✓ *You can play with yourself.*
- ✓ *You can play with your own toys, but you can't take them apart*
- ✓ *You can play with toys that were given to you.*
- ✓ *You can play with toys you've made yourself.*

Benefits of Demeter's Law

- Coupling Control
  - reduces data coupling
- Information hiding
  - prevents from retrieving subparts of an object
- Information restriction
  - restricts the use of methods that provide information
- Few Interfaces
  - restricts the classes that can be used in a method
- Explicit Interfaces

Acceptable LoD Violations

- If optimization requires violation
  - Speed or memory restrictions
- If module accessed is a fully stabilized “Black Box”
  - No changes to interface can reasonably be expected due to extensive testing, usage, etc.
- Otherwise, do not violate this law!!
  - Long-term costs will be very prohibitive