PROGETTAZIONE DI SISTEMI ORIENTATI AGLI OGGETTI (OOD)

Michele Marchesi

michele@diee.unica.it

Dipartimento di Ingegneria Elettrica ed Elettronica

Università di Cagliari

Agile OOA & OOD

- The analysis phase has the goal to understand and specify system requirements.
- This is accomplished building models of the system that are then used as starting point for the design of the system architecture.
- The main issues of OOA are:

1. Understanding the system to be built.
2. Managing the complexity partitioning the system into subsystems.
3. Documenting the work done, in such a way that this documentation helps understanding the system.
4. Developing models of the system, also using graphic notations, that will ease system design and coding.
5. Formally specify the system, enabling to write a binding contract for customers and developers (let's take this apart!).

Agile OOA & OOD

- The first four issues deal with intermediate steps from requirement elicitation to system implementation.
- These steps do not add tangible value to the customer, in the sense that OOA does not directly yield working software.
- So, OOA is a transitory phase, essential to system development but not a goal to itself.
- OOA analysis techniques:
  - Incremental delivery based on feature implementation. Add more and more features, in short iterations.
  - Interactive analysis using CRC method, maximizing communication. Do not record analysis results, but put them into working code as soon as possible.
  - Spike solution (disposable working prototypes).
  - The main documentation medium is code, which must be written in proper, self-documenting style. UML drawings should be added only if the benefit of having them is greater than the cost of creating them and keeping them aligned with the code.

Agile OOA

- In a nutshell, agile development techniques obviously do perform analysis, but:
  - they do not try to perform an up-front complete analysis of the system;
  - they get feedback from the customer on the validity of the analysis and design, implementing the analysis models as soon as possible, also with the help of throw-away prototypes, if needed;
  - they do not consider analysis models and documentation a goal in itself, but keep and maintain them only if really needed.

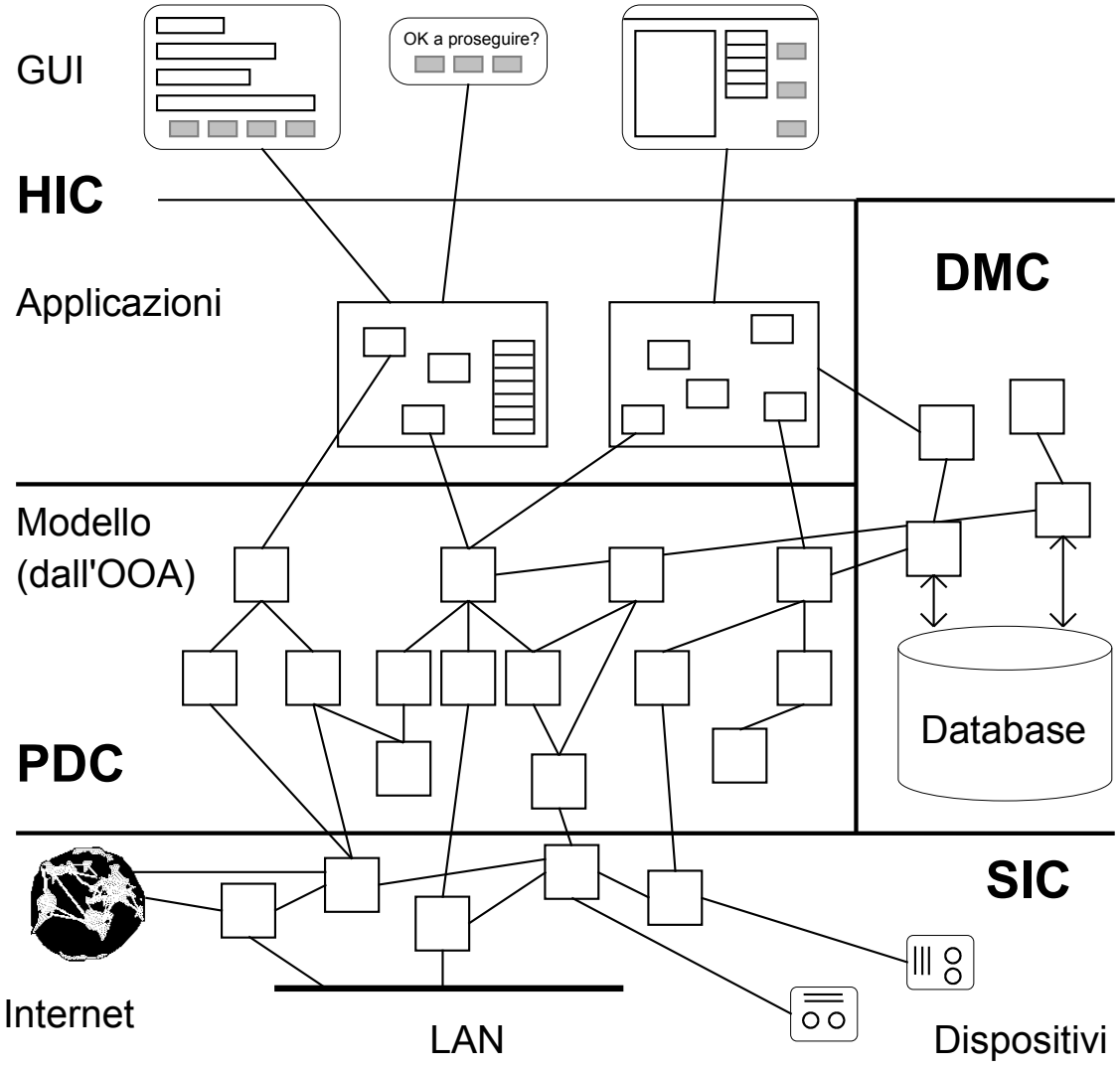
OOD Main Subsystems

The typical sub-systems of an OO software application are:

- **The object model**: the kernel of the system, where key processing takes place. It is directly derived from OOA model. With respect to OOA, the OOD model gives details on relationships implementation, interface specification and data structures.
- **The human interaction component (HIC)**: the objects of the user interface. These objects are the windows and the widgets of the GUI, and the applications able to take user inputs, to run the requested processing and to show the results to the user. These applications are themselves objects, able to access and use the objects of the model.
- **The data management component (DMC)**: the objects whose responsibility is to permanently store and to retrieve the data.

The system interaction component (SIC): the interface with external devices, network and the Internet.

OOD Main Subsystems



Key OOD principles

Abstraction

- The world is very complex, and that every object in the world is deeply linked to many other objects.
- If we would consider in detail every aspect of a single “thing” of the world, including interrelated things and without limitations, we would end to consider the whole universe!
- This would be perhaps sensible for a mystic, but it would impede the development of any science.
- Abstraction is the tool enabling us to overcome this problem.
- Applying abstraction, we consider only the aspects of an entity that are relevant to the problem at hand, neglecting the others.
- A system, or a module, may be viewed at various *levels of abstraction*. At the highest levels, only the key, synthetic details are considered, at lower levels more details can emerge, specific of the aspect under consideration.

For instance, a train...

Abstraction	**The object *Train* is seen as:**
Highest level	A train represents in the system a real train, tracked by the system or simulated.
Lower level, focusing on data structure	A train has an unique identifier, knows its maximum speed, its typical acceleration and deceleration, the track section it is positioned, etc.

Lowest level, focusing on data structure	Data structure of a train (instance variables):
	`string trainId;  // Unique identifier of the train`
	`float maxSpeed;  // Maximum speed in Km/h`
	`float acceleration;  // Typical acceleraition in m/s`^{2}

Lower level, focusing on permanent memorization	The data of the trains are stored on a file in a given format.
Lower level, focusing on simulation	The train knows the track section its head is positioned on. The train can compute the time it takes to cover a given length, starting from its current speed. …
…	…

Scale

- The concept of *scale* is related to levels of abstraction.
- A system can be viewed at different scales, like a geographic map.
  - A system can be represented at the highest level as a set of sub-systems exchanging information among them.
  - At a lower level, these sub-systems can be expanded, considering the sub-sub-systems they are composed of.
  - Then, each of these sub-sub systems can in turn be expanded, showing the modules it is composed of.
  - Eventually, each single module can be expanded and represented, for instance in form of public interface and internal representation.
- When applying scale, we consider the same aspect of the system, but at different abstraction level.
- The notations used to graphically show the analysis or design of a system usually allows to look at the system at different scales, thus facilitating understanding the underlying model.

Iterative Refinement

- *Iterative refinement* is a design strategy originally proposed by Wirth [1971] in the context of procedural programming, but which can be applied also to OOD.
- In the original definition, a program is implemented by successively refining levels of procedural detail:
  - We start from the *main* procedure that is the "top" of the whole system, or with a top level procedure to implement.
  - Using procedural abstraction, we decompose this procedure into more detailed instructions and lower-level procedure calls.
  - These will be decomposed in the next step, and so on.
  - The successive decomposition of specification terminates when all instructions are expressed in term of the underlying programming language.
- As procedures are refined, so the data they operate on may have to be refined
- Every refinement step implies some design decisions, considering alternative solutions.

OO Iterative Refinement

- In an OO system, iterative refinement may be applied both to class hierarchy definition, and to method design and implementation.
- In the definition of class hierarchies, we start considering the higher classes in the hierarchy, for instance class *Person* in a hierarchy of human roles.
- When the need of more specialized classes appears, we add them to the hierarchy, properly restructuring the attribution of data and methods, and so on iteratively adding more and more specialized classes.
- For instance, an *Employee* and a *Customer* class could emerge, adding specific behavior to a generic *Person*. Then, a *Manager* and a *Secretary* classes could add behavior to *Employee*, and *TopManager* to class *Manager*, an so on.
- The OO system is also based on methods at different detail levels, and the design proceeds from the highest level methods to the lower level one, following iterative refinement.

Information Hiding

- A module must hide its internal implementation from all other modules.
- This principle, to be followed when designing a module, was introduced by Parnas [1972] and is called information hiding.
- A module must be accessed only through its public interface, and this access should depend in no way on its implementation.
- This means that changing the implementation of the module without changing its interface should have no effect on the rest of the system.
- No direct access should be provided to the internal data structure, or to the private procedures of a module.
- This prescription is often violated for performance reasons; whenever possible, other ways to improve performance should be found.

- Applying information hiding, we use abstraction to

define the modules and their interfaces.

- The access to internal data and procedure is strictly controlled, and may be used also to define and enforce consistency constraints.
- For instance, it is easy to enforce the constraint that a data must always be strictly positive, if there is only one public function entitled to change the value of this data.
- Information hiding is a principle aimed to clearly define the boundaries of a module for external access.
- However, it should be used as far as possible also in the internal implementation of the module.
- For instance, accessor methods (setter and getter)

Implementing Information Hiding

Principle 1. Access instance variables of objects only through accessors.

- In this way:
  - Data consistency can be verified and enforced by setters just in one point.
  - Getters may return copies of the data, and not the data themselves, to enforce security.
  - If the data structure changes, only the accessors should change. The other methods do not require to change.
  - Computed values can be seamlessly used as if they were actual instance variables.
  - Data access can be controlled with respect to locks.
  - Data access can be logged for debugging.

Principle 2. Minimize the number of public and protected methods.

- We remember that
  - only public methods may be called from outside a class;
  - only public and protected methods may be called from subclasses' methods,
  - private methods may be called only inside methods of the same class.
- A class should perform a single, clean task, exposing only the essential behavior to external world.
- The smaller the number of its public methods, the looser its coupling with other parts of the system.
- A class with a few public methods is easier to understand and use. Moreover, in the case of changes to it, it is easier to keep their interface unchanged.

Principle 3. (Law of Demeter) An object O in response to a message M (that is, executing method M) should send messages only to the following receiver objects:

1. ***O itself,***
2. ***objects sent as arguments of message M,***
3. ***new objects O creates while executing method M,***
4. ***objects which are directly accessible instance variables of O (got using getters),***
5. ***objects which provide global services to O (global variables).***

- A consequence is that local variables of a method may only hold new objects created while executing the method: inside a method, you should not obtain objects but those referred to in Demeter's Law.
- Methods should operate only on objects directly available to them, and should not obtain intermediate object to send them messages. This is also known as the "kill the middle-man" principle.

Styles of reuse

- Composition:
  - Additive
  - Projective, when the purpose of the new object is to wrap an existing object in order to hide some behavior and expose others.
- Schemata: general structure, parameterized using one of more abstract classes, able to generate a specific instance
  - *templates*
  - *interfaces*
- A third fundamental reuse mechanism, peculiar to object-oriented systems, is inheritance.
- With inheritance, it is possible to use *polimorphism*:
  - send a message to an object
  - the object will call the correct method, depending upon its class

Principles regarding inheritance

Principle 4. (Liskov's Principle of Substitution) If a class B is "just like" a class A except for extensions, then it should be possible to use a B object anywhere you an use an A object. That is, a child (subclass) should be able to be used where ever the parent (superclass) can be used.

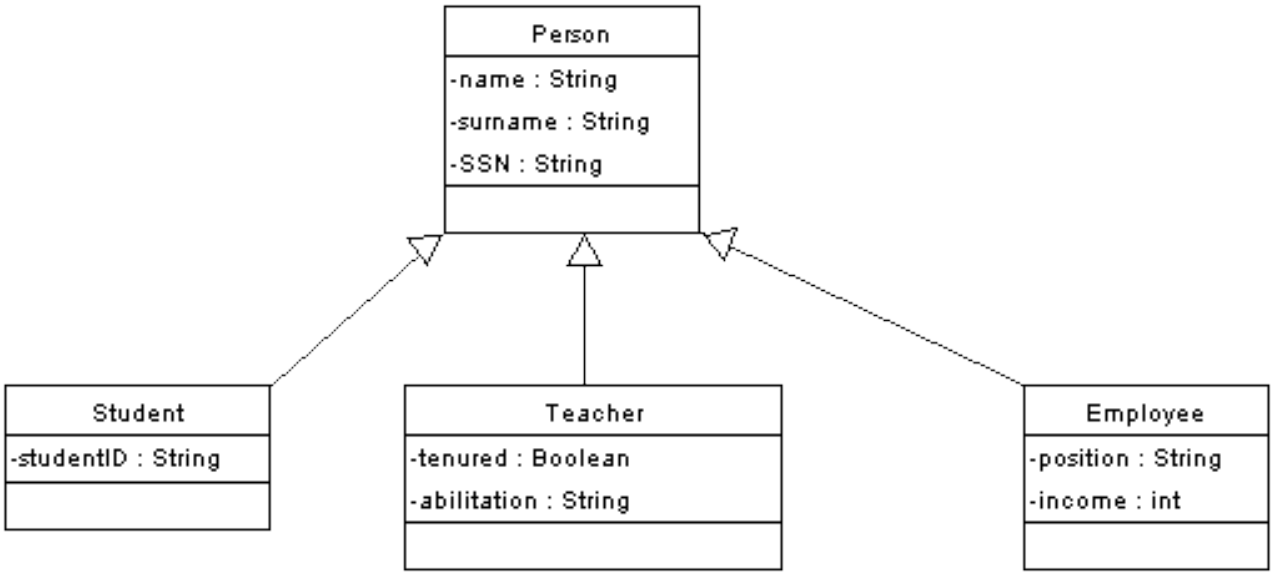
Design your classes to preserve this property unless you have a strong reason to do otherwise.

Principle 5. Use inheritance only to reuse and extend functionalities. Use inheritance only if it reflects a relationship existing in the real-world.

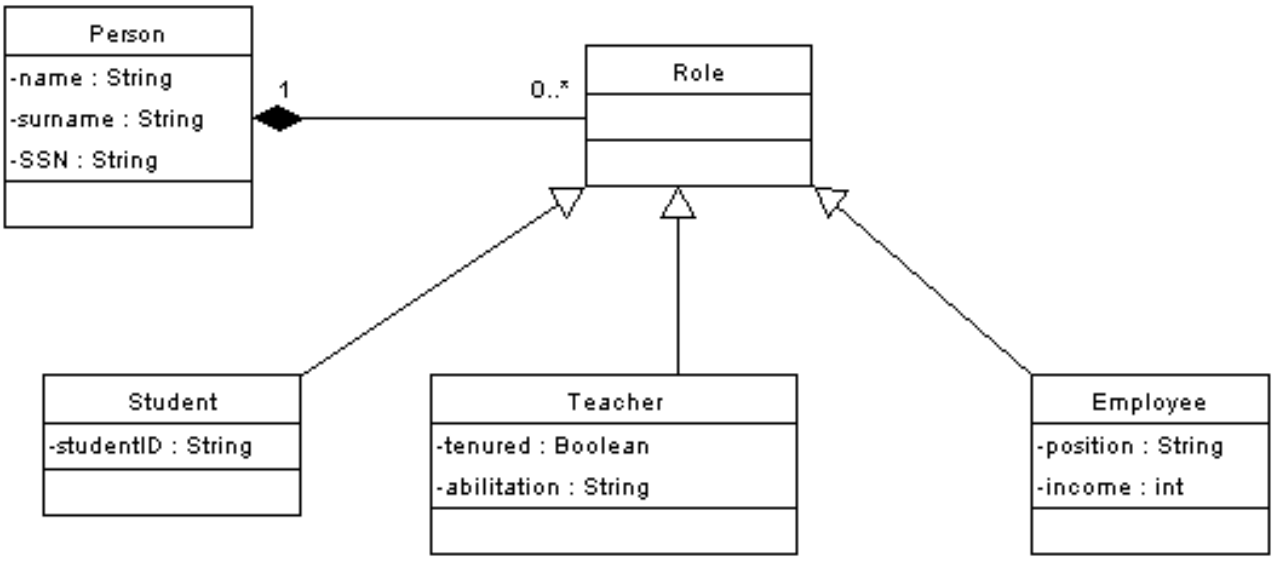
- Inheritance should not be used to reuse implementation (code). It should only be employed to reuse and extend functionality. To reuse implementation, use instead composition.

Roles

- An exception to Principle 5 is the case of roles.
- Typical examples of roles are human roles: in a given system, a person can play different roles.
- For instance, in a university management system we might deal with a graduate student who teaches courses to undergraduate students, and who is also a part-time employee of the university
- The most straightforward way to design an object-oriented model of this part of the system is to acknowledge that Student, Teacher and Employee are all kinds of Person:



- This model reflects the real world, and follows Principle 5.
- However, in this framework it is not possible to easily accommodate our student, who is also a teacher and an employee.
- Three objects might be created, one for each subclass, to keep track of him/her. This would replicate three times his/her data derived from Person, and there would be no explicit information that the three objects refer to the same person.
- The use of roles solves the problem:



- This also reflects the real world, since we may say that "being student (or teacher, or employee) is a role of person".

Principle 6. Roles are acquired via composition, not by subclassing.

- Composition should also be used in place of inheritance by exception:
  - If a new class B is like an existing class A, but with restrictions on its behavior, then B should not be derived from A via inheritance.
  - Instead, class B should contain an instance of class A (composition).
  - The part of B's behavior that is correctly implemented by A is directly delegated to A.
  - The remaining part of B's behavior is suitably re-implemented by B.

Low Coupling

- Two modules are highly coupled when there are many dependences between them.
- Two modules are loosely coupled if interconnections and dependences are weak.
- Two modules are uncoupled if there is no interconnection, and they are independent.
- There are many ways the modules can be dependent on each other.
- In **increasing coupling order**, they are:
  - The procedures or methods of a module calls procedures or methods defined in other modules.
  - The procedures or methods of a module have parameters, explicit or implicit local variables or return types defined in other modules.
  - A class defined in a module is subclass of a class defined in another module.
  - One or more modules make calls to specific API procedures of the operating system.

- One or more modules are interfaced to specific devices, or make use of specific data formats and communication protocols (I/O coupling).
- Two (or more) modules access a shared area containing global variables (common coupling).
- Two (or more) modules directly access data contained in another module (content coupling).

- The two last kinds of coupling are the worst, and should be avoided.
- Content coupling clearly does not satisfy the information hiding principle.
- The other forms of coupling are unavoidable, since a system composed only by completely independent modules would be in fact a set of different systems.
- Sub-systems and modules, however, should be defined in such a way to minimize (the unavoidable) coupling.

Why minimizing coupling

- The reasons to minimize coupling are two:
  - A self-contained module, making minimum use of other modules, is simpler and easier to implement, understand and maintain.
  - If the design and implementation of one or more modules have to be changed, the impact of such a change to other modules of the system is kept minimum.

Cohesion

- A module is cohesive if it performs a single task and if all its elements are directed toward and essential to this task.
- As with coupling, cohesion may be represented as a "spectrum". We should strive to have high cohesion in every module, but this cannot always be accomplished.
- Kinds of cohesion (from the lowest to the highest):
  - Coincidental: a module composed by many unrelated parts
  - Logical: a module is composed by logically related parts, but with no other interactions among them. For instance, a module aimed to manage all forms of outputs
  - Temporal: elements are grouped into a module because they are all processed within the same limited time period.
  - Procedural: a module is composed by parts in which control flows from one activity to the next.

For instance, a module providing access to Web services, and performing all the related tasks.

- Communicational: all elements of a module operate upon the same input data set and/or produce the same output data.
- Sequential: the activities of the module are arranged such as the output of one serves as the input to the next.
- Informational: multiple functions of the same module have access to the same data structure or resource that is hidden within the module.
- Functional: the module performs exactly one action, or it achieves a single goal. All parts of the module contribute to just one function, nothing else is in the module and everything needed for the function is in the module.

- Only the last two kinds of cohesion are good, while sequential cohesion may be acceptable.
- All other forms should be avoided

OO Cohesion

- For OO systems, we can still define cohesion, and strive for high cohesion.
- Here we should substitute "class" for "module".
- A class with the weakest forms of cohesion is a collection of methods, that alone or together exhibit one of the weak forms of cohesion.
- A common design behavior that yields non-cohesive classes is to use multiple inheritance to give to a class features that are not pertinent with its main scope.
- A class is functional cohesive if it represents a single concept, and each operation in its public interface is functional cohesive.
- OO designers should strive to create designs with classes having maximum cohesion.

Simplicity

- Simplicity of design is a quality easy to state, but really difficult to formalize and achieve.
- Paraphrasing a famous Einstein's sentence, your design should be as simple as possible, but not simpler.
- Simplicity has to do with achieving the goals of overall design, and the goals of every module, with the minimum number of artifacts, responsibilities, collaborations, variables, operations, number of instructions.
- Usually, simplicity is obtained at the end of a long trial-and-error process, using refactoring

> ***Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius -- and a lot of courage -- to move in the opposite direction.***

How to achieve simplicity

- Simplicity may be achieved at various levels:
  - At method level, writing methods short and with small signatures.
  - At module level, keeping the public interface of the module as small as possible.
  - At system level:
    - holding the information in the modules that need it, avoiding “middle-man” modules that only pass information from one module to another, without performing useful processing;
    - minimizing the information and control paths between modules;
    - avoiding global data areas and global variables;
    - keeping the inheritance hierarchies as small as possible, and putting the information and operations at the right hierarchy level.
  - At every level, **avoiding code duplications.**

- The main guideline for obtaining simplicity is:

> ***Build only the code that you need to satisfy your present requirements, without trying to anticipate future needs and requirements.***

- The right design for the software at a given time is one that

1. Runs all the tests.
2. Has no duplicated logic.
3. States every intention important to the programmers.
4. Has the fewest possible number of classes or methods.

> ***Perfection is reached not when there is no longer anything to add, but when there is no longer anything to take away. (A. Saint-Exupery)***

Infrastructures

- A common practice in software development of new systems is to spend the first part of the project designing and developing an infrastructure — frameworks, services and modules that will be useful to develop more quickly the system.
- Examples of infrastructure are a framework to build dynamic Internet pages, or a framework to ease software localization, i.e. writing different versions to be used in different languages.
- Unfortunately, this practice has three drawbacks:
  1. It does not yield immediate value to the customer.
  2. Often, it takes too long.
  3. Most likely, many of the infrastructure's features will never be used.
- In OOD, building such up-front infrastructures should be avoided.

- Do not build infrastructure components and frameworks for the first case that needs a functionality.
- The second time you need it, you will extract the component yielding that functionality, to follow the no-duplication principle.
- In this way, the needed infrastructure will grow as it is needed, but not more, without delaying delivering value to the customer.
- Sometimes, it is obvious that an infrastructure must be developed up-front. In this case, follow these tips:
  - Carefully scrutinize the problem, to be absolutely sure that the infrastructure actually needs to be built up-front.
  - Keep the infrastructure's features to a bare minimum. New features will be easily added when needed.
  - Do not delay producing software able to give immediate value to your customer.

Design By Contract

- Design by contract is a technique that provides a rigorous specification of each method of a class, and the class's legal state, first introduced by Bertrand Meyer.
- For each significant method of a class, the designer should state preconditions and post-conditions.
- **Preconditions** are assumptions about inputs to the method, and the object state just before the method is executed, that must hold for the method to work properly.
- **Post-conditions** are assumptions about outputs of the method, and the object state just after the method is executed, that are guaranteed to hold true.
- Developing software is like a contract between the developer of the method and the developers of the software that calls such a method.

- Callers must guarantee that preconditions are satisfied when the method is called. If this is true, the method's developer guarantees in turn that post-conditions are satisfied.
- If preconditions do not hold, the method is authorized not to work properly, and even to crash the system – the caller has not honored the contract.
- If preconditions hold, and even one post-condition is not satisfied, the developer of the method may be pleaded guilty, since he did not honor the contract.
- An example: the square root function:

```
double sqrt(double x)
```

- This function computes the square root of a number.
- Its precondition is that the parameter x must be a real number, and must be $x >= 0.0$.
- Its post-condition is that the function must actually compute the square root of its argument, within the precision of the computer:

$$abs(x - sqrt(x) * sqrt(x)) <= \varepsilon$$

- Another powerful tool in design by contract are **assertions** and **invariants**.
- They are specifications of legal state of the various classes of the system. They may include constraints on data values and a requirement that the values represent what they are intended to represent.
- Design by contract is a powerful design tool, particularly useful in the development of large systems.
- It helps to explicitly state assumptions about code that often are left implicit, hindering communications and causing misunderstandings among developers.

Metaphor

> ***"The system metaphor is a story that everyone (customers, programmers, and managers) can tell about how the system works" (Kent Beck)***

- The Metaphor is a design practice targeted to define in an unconventional way the OO Architecture, that is the key classes and objects of the system, and how they interact.
- The team and the customer agree on a common system description, and a common "system of names" that guide development and communication.
- The metaphor must be easily understood by both the developers and the customer.

Reasons to seek a system metaphor

- **Common Vision**: The metaphor suggests the key structure of how the problem and the solution are perceived. This can make it easier to understand what the system is, as well as what it could be.
- **Shared Vocabulary**: The metaphor helps suggest a common system of names for objects and the relationships between them.
- **Generativity**: The analogies of a metaphor can suggest new ideas about the system.
- **Architecture**: The metaphor shapes the system, by identifying key objects and suggesting aspects of their interfaces. It supports the static and dynamic object models of the system.

Finding the “right” metaphor

- For each system, the baseline is the **naïve metaphor:**
  - ***let objects be themselves!***
  - For instance:
    - a payroll system might have Employee, Union, Pay and Check objects
    - a race simulator might have car, driver, race...
  - In this case, customer and developers agree that the names of the system will be taken from the problem domain
- The naïve metaphor is enough when both customer and developers have a fair, common understanding of the problem domain.
- On the other hand, when a metaphor is needed? The answer is when there is no shared vision of the system between customer and developers, and the metaphor can yield this vision.

Refactoring

> ***"Perfection is attained by slow degrees; it requires the hand of time". (Voltaire)***

> ***"Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" (Martin Fowler)***

- It does not alter the external behavior of the code. Refactoring is not made to add features to the code.
- It improves code's internal structure. Refactoring is made to make the code simpler, easier to read, better organized, easier to modify.
- Refactoring is not just cleaning code. Code is changed (and cleaned up) in a more efficient and controlled manner.

The main benefits of refactoring

- Refactoring improves the design of software:
  - If refactoring is continuously applied to a software system, its architecture gradually improves, and a good design emerges.
  - Another reason to refactor is because systems are always subject to changes. If these changes are made only locally, the code loses its structure, and in a short time the cost of changes become unmanageable. Applying refactoring brings again order to the system.
- Refactoring makes software easier to understand, being aimed also to increase system readability.
- Refactoring helps to find bugs
  - Refactoring is a kind of software inspection.
- Refactoring in the long term helps your productivity

When to refactor

- During software development, you don't allocate a given time to refactor.
- You refactor when you find some signals in the code, that tell you it is time to refactor.
- You are able to find such signals when you. This happens when you are are reading the code adding new features to the code, fixing a bug, or performing code reviews.
- The signals in the code that should trigger refactoring are the followings:
  - Presence of bugs.
  - When it is difficult to add a new feature to the system.
  - When code is obscure.
  - When performances must absolutely be improved.
  - "Bad smells" in the code.

Bad smells in the code

Too much code:

- **Duplicated Code. The most important bad smell.**
- **Large Method**. As a rule of thumb, any method longer than 10-12 lines of code (in Java and C++), should be scrutinized.
- **Large Class.** The rule of thumb is that a class should not have more than 8-10 instance variables, and 10-20 methods, besides accessors, constructors and destructors.
- **Data class**. Data classes are containers for data. Here, the rule of not having more than 8-10 instance variables does not apply.

> ***"Data classes are like children. They are okay as starting point, but to participate as a grownup object, they need to take some responsibility" (Kent and Fowler).***

- **Refused Bequest.** A subclass does not need methods inherited from its superclass
- **Feature Envy.** A single method sends many messages to the same object.
- **Temporary Field.** An instance variable is only used in some circumstances, depending on the value of another variable.
- **Strikingly Similar Subclasses.**
- **Expensive Set Up.** The constructors, or the initialization methods of an object are very complex.
- **Unused Code**. If there are methods, or even classes, not actually ever used, wipe them up.

Not enough code:

- **Lazy Class**. Small classes with too few instance variables, or with too little code
- **Incomplete Library Class**. Do not fear to modify the library or the framework, adding the required classes or methods.

Not actually the code:

- **Comments**. The need of many comments to make the code understandable is a symptom of bad code.
- **Excessive Logging**. Lots of logs are needed to figure out what the code is doing.

Problems with the way the code is changing

- **Divergent Change**. A class usually needs to be changed in many points every time you need to change it.
- **Shotgun Surgery**. Changes are made to many classes. (I.e.: parallel inheritance hierarchies).

Other code problems:

- **Data Clumps**. If you see the same handful of data items together, in many places, consider to create a class holding them.
- **Switch Statements**. The presence in the code of "switch" statements is a very bad smell.
- **Message Chains**. If in a method a message is sent to an object to obtain another object, to which a message is sent to obtain yet another object...
- **Middle Man**. A class systematically delegates to another class a large percentage of its behavior. Many methods compute their result sending a message to the same object, and returning its result.
- **Inappropriate Intimacy**. A class makes direct access to private variables and methods of another class. The "friend" keyword of C++.
- **Alternative Classes with Different Interfaces**. Methods that do the same thing in different classes, do not have the same name and signature.

- **Same Name Different Meaning**. Methods, in different classes, with the same name but with different meaning.
- **Law Of Demeter Violations**.
- **Long Method Names**. Long method names are often an indication that the method is in the wrong class.
- **Embedded Code Strings**. Large chunks of SQL, HTML or XML embedded in your code are difficult to understand and maintain.

Kinds of refactoring

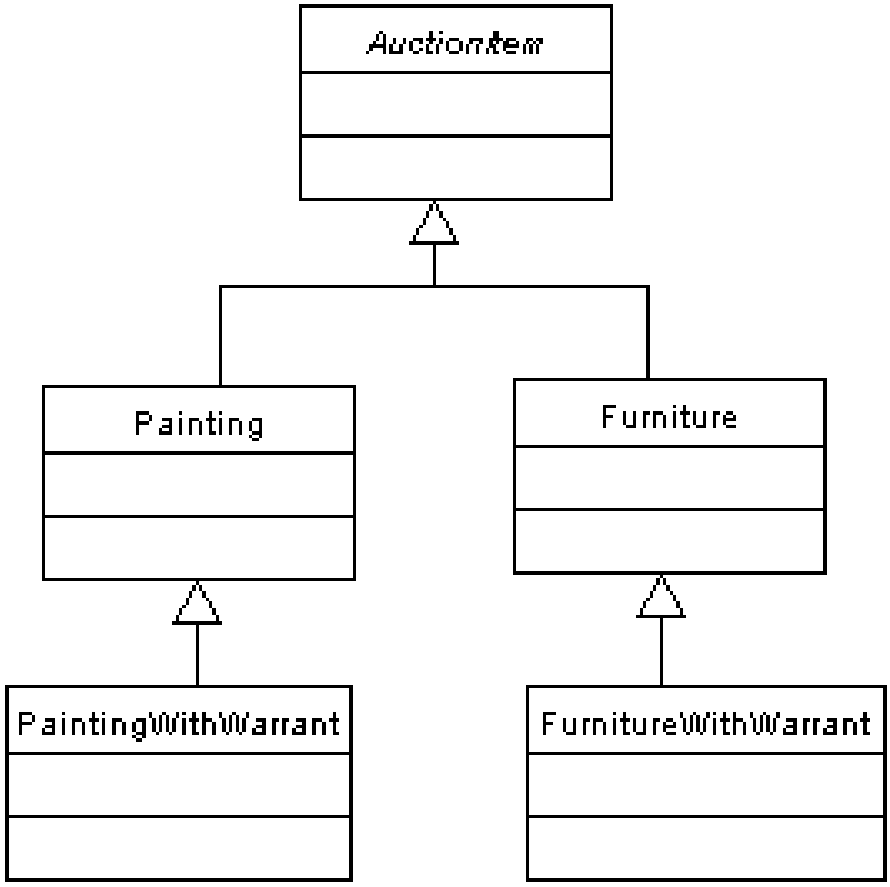
- See Fowler's Refactoring book [Fowler 1999].
- **Composing Methods**: The simplest and easiest form of refactoring is changing the structure of a method. These refactorings involve extracting a method from one or more, inlining a method, tiding a method acting on its parameters or temporary variables
- **Moving Features Between Objects**: move methods or instance variables between classes, extract a class from an existing one, inline a class. It is about increasing the cohesion of classes, reducing coupling, and putting behavior close to the data.
- **Organizing Data**: creating new objects, in place of scattered fields or other structures.
- **Simplifying Conditional Expressions**

- **Making Methods call simpler**: renaming a method; adding or removing a parameter to/from a method; replacing a parameter with a method call; replacing error code with explicit throw of an exception.
- **Dealing with Inheritance**: pulling up or pushing down the hierarchy an instance variable or a method; extract a superclass, a subclass, or an interface; replacing inheritance with delegation, or vice-versa.

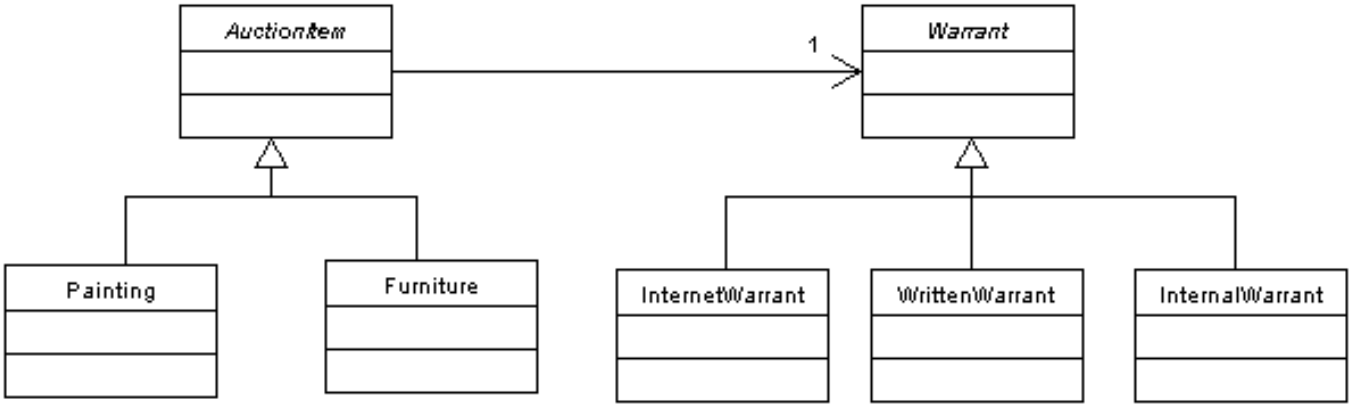
Big Refactorings

Tease Apart Inheritance

- This refactoring deals with a tangled inheritance hierarchy that tries to combine different features in a confusing way.
- For instance:



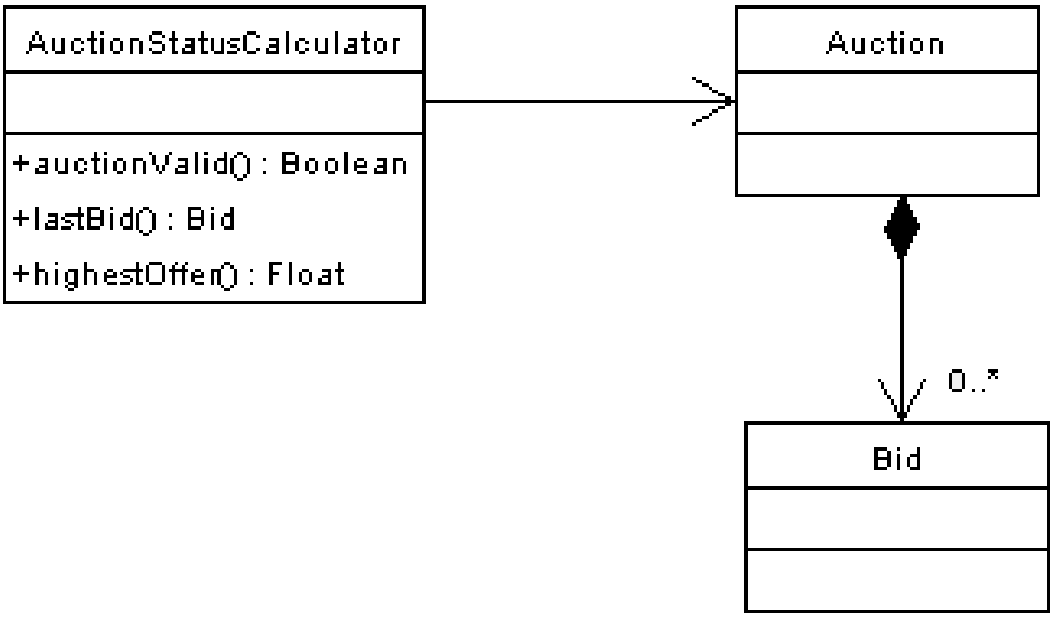
- After the refactoring:



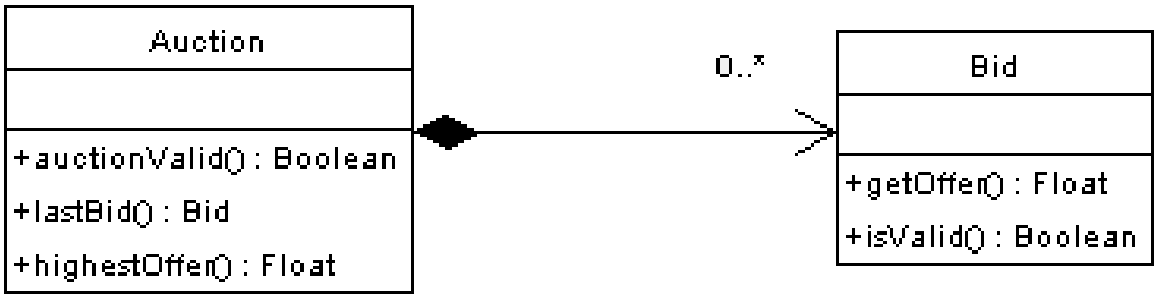
- Tangled hierarchies are quite common, especially when OO designers are inexperienced, or come from procedural design.
- Realizing when a hierarchy is trying to do two, or more, jobs, and refactoring it to a cleaner structure ensures a more understandable and simpler program.

Convert Procedural Design to Objects

- This refactoring has to do with “procedural object”, which try to do everything, using other objects as a mere data structure.
- Classes whose name includes “manager”, “calculator”, or end with “er” are natural candidates to be scrutinized for this kind of refactoring.
- An example: “Auction” and “Bid” classes, only act as structures (“struct”) holding their data. An “AuctionStatusCalculator” class reads these data, computes and returns the required information.



- The alternative, object-oriented way, to perform the same computation:



- Here we get rid of the procedural object, and put the computations close to the data they operate on, that is in classes “Auction” and “Bid”.

Separate Domain from Presentation

- Often, the easiest way to write GUI applications is to embed in the code of the GUI class all the behavior of the application.
- This programming style is encouraged by many IDEs, adopting a logical two-tier design:
  - the data are stored in the database
  - the logic is coded in the presentation classes
- This situation should be refactored, separating the domain logic from the presentation logic.
- The domain classes should contain no visual code, but all the business logic, and should not be aware of presentation classes operating on them.
- The GUI classes should contain only the logic to deal with the user interface.
- This approach facilitates future changes both to the business and the presentation logic, and also allows multiple presentations of the same business logic.

Extract Hierarchy

- This refactoring should be made when you have a class whose methods make extensive use of conditional “switch” statements.
- The use of “switch” statements is not OO, but should be substituted with polymorphism.