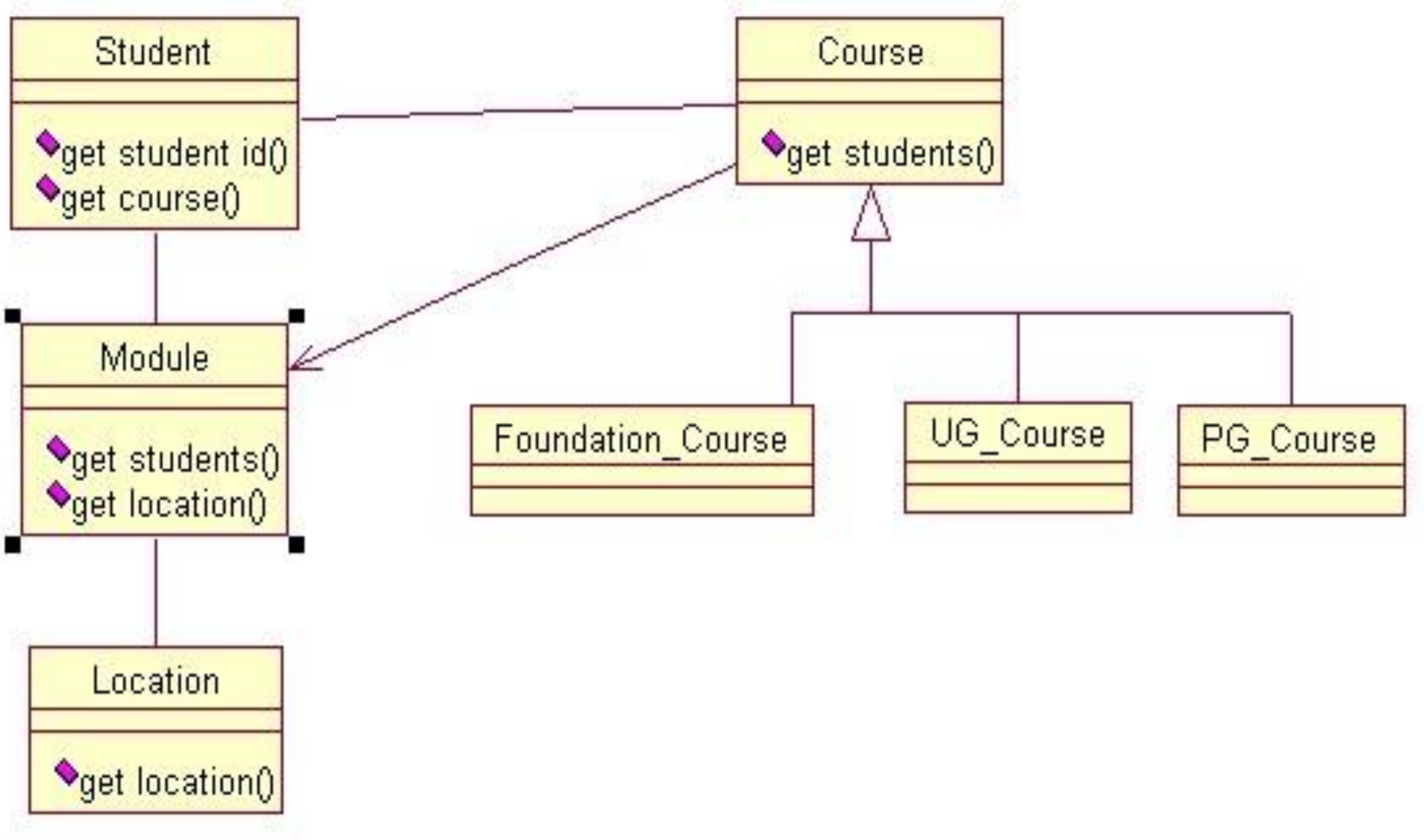
7SENG003W Advanced Software Design

Design Heuristics + SOLID

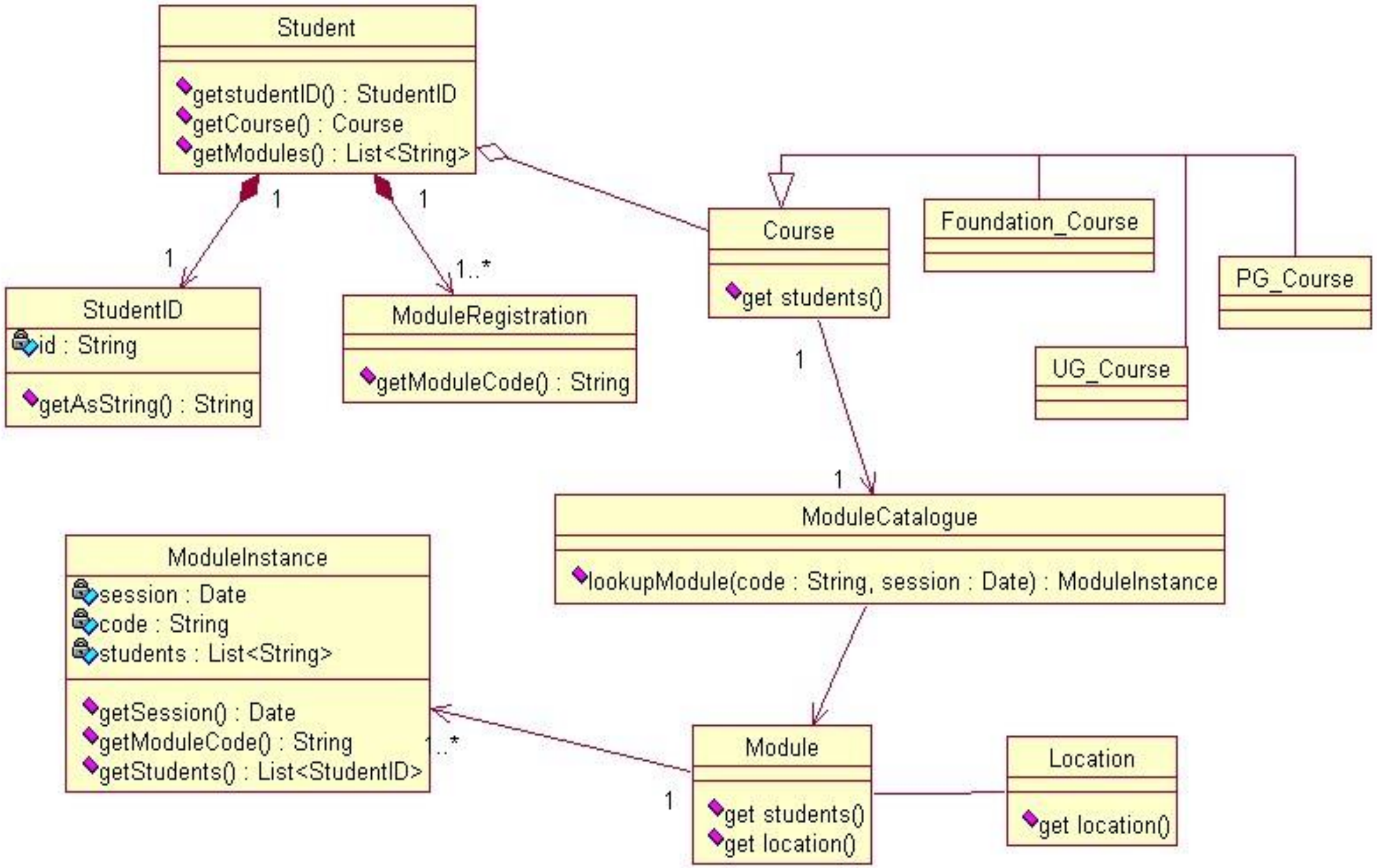
Design

- So far:
  - Use Case analysis
    - Use case model: Use case diagram and Scenarios
  - Analysis
    - Analysis model : Sequence and Class diagrams
- Now design starts - where we start designing the actual system
- The design stage takes the output from the analysis model and refines it to the point where it can be turned into code
- Typical refinements: design heuristics / design patterns

Refinement we want to go from...



.. towards a more detailed model



ABSTRACT → DETAILED

- Going from a high-level analysis model to a low-level detailed model will involve making decisions
  - About operations (which parameters and return types)
  - About attributes (what concrete types etc.)
  - About classes
- The decisions we make about classes are probably the most important. We need to consider:
  - Whether current class structure is efficient
  - Whether it promotes things such as code reuse
  - Whether it is easily maintainable
  - Etc.
- For example: move from Student → Module relationship to Student → ModuleRegistration to break dependency between Student and Module – good idea?

“Good” Design

- Usually there will be a choice of different designs for any particular problem
- Are they are equally “good”
  - Does it matter what classes we have or how they are related, so long as the software “works”
- If the answer is “no”, then all designs are equally good (and “good” has no effective meaning)
- If the answer is “yes”, then some designs are better than others
  - So we need to recognise good design
  - We need to know what we mean by 'good'
  - How do we create good designs?

Definition of 'good' design

- Coad and Yourdon (“Object-Oriented Design”)
  - “A good design is one that balances trade-offs to minimise the total cost of the system over its entire lifetime”
  - Where costs include:
    - Costs of analysis/design
    - Costs of coding
    - Costs of testing/debugging/maintenance etc.
- Meyer (“Object-oriented software construction”) identifies several areas of “internal quality”, such as:
  - Correctness
  - Robustness
  - Extendibility
  - Reusability
  - Compatibility

“Good” is based on experience

- Software design community has learnt the hard way what 'good' means – through failures and mistakes, and working out why these happen...
- No formal way to ensure that a design is good, but software design community has realised that:
  - There are some high-level abstract notions that help us design good systems – encoded as design principles or heuristics
  - There are some generally applicable design solutions that can be used in a wide range of different contexts and which encapsulate best practice – design patterns

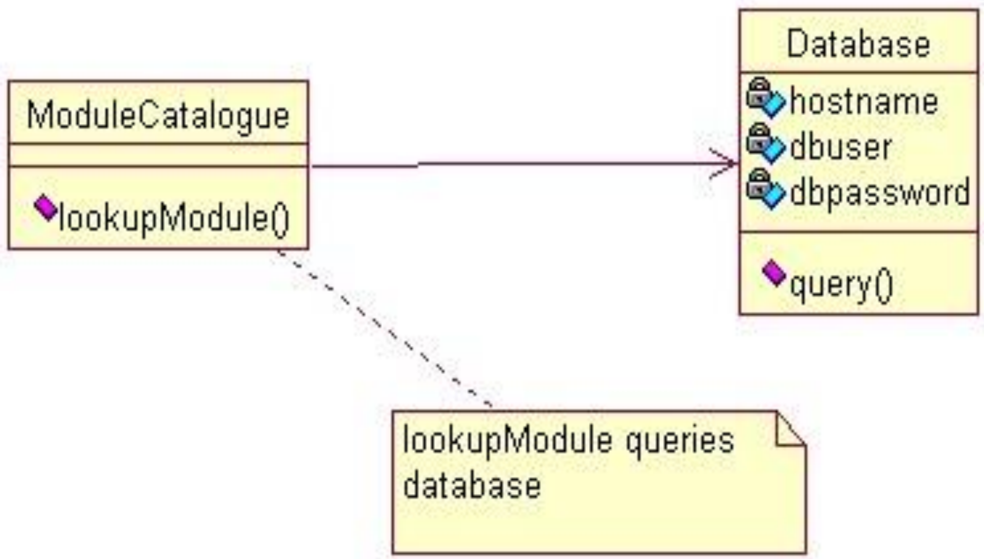
What is a Design Heuristic

- A heuristic is a general principle or rule of thumb that experience has shown helps improve a design
- Examples:
  - Class interfaces should be complete and minimal (Scott Meyers, Effective C++ Design)
  - A class should capture one and only one key abstraction (Arthur Riel, OO Design Heuristics)
  - Keep related data and behaviour in one place (Reil)
- Most heuristics deal with the consequences of coupling and cohesion
- Some are more formal than others, but all aim at improving quality of design

Coupling

- Describes the degree to which one component **depends** on another.
- We aim for loose coupling, to reduce dependencies between classes
- Tight coupling means that different components are highly dependent on each other
- Loose coupling means the degree of dependence is low
- Dependence is measured in terms of how changes in one component affect another component
  - We can't eliminate coupling completely – that would mean objects not connecting to each other
  - But we want to reduce the probability that changes in one thing mean changes in another

Tight coupling example



- Simple example: ModuleCatalogue depends directly on the interface of Database class.
- The code in lookupModule may use the methods of the Database class – so any changes in this interface may cause changes in ModuleCatalogue

Code example

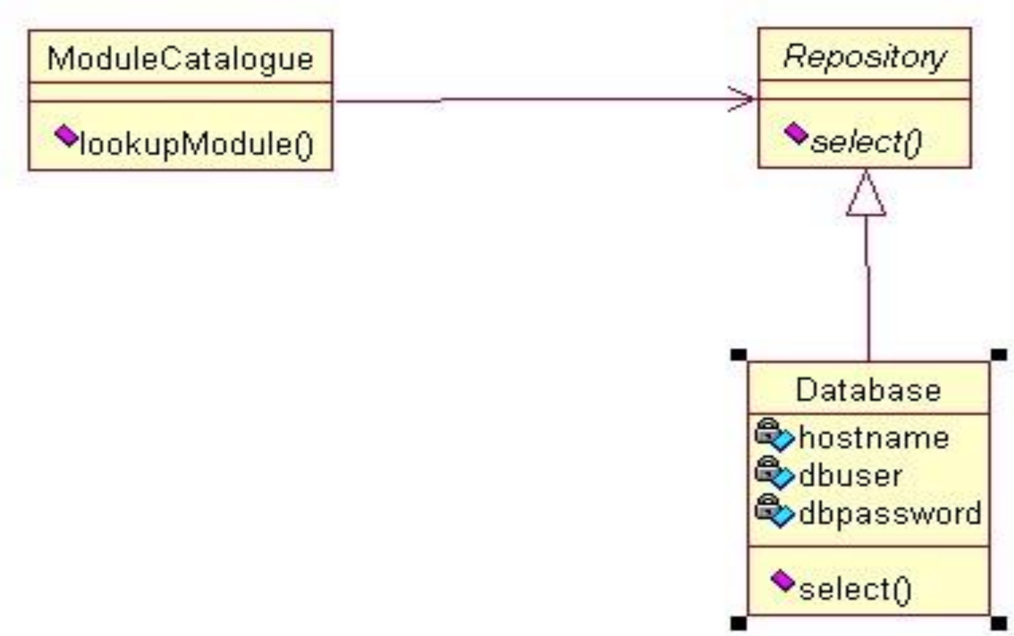
```
class Database {
    private string hostname;
    private string dbuser;
    private string dbpassword;
    private string dbname;

    Result query(string query)
    { /* make query against db and return results */ }
    // ... rest of class
}

class ModuleCatalogue {
    private Database db;

    public string lookupModuleName(string moduleCode) {
        Result r = db.query($"SELECT name FROM module WHERE code={moduleCode}");
        // do something with Result and return module name
    }
}
```

If Database::query function changes its interface, this will cause changes to code of ModuleCatalogue::lookupModule

Changes to one class cause changes to tightly coupled classes

```
class Database {
    private string hostname;
    private string dbuser;
    private string dbpassword;
    private string dbname;

    public Result ResultSet { get; }

    void query(string query)
    { /* make query against db and return results */ }
    // ... rest of class
}

class ModuleCatalogue {
    private Database db;

    public string lookupModuleName(string moduleCode) {
        db.query($"SELECT name FROM module WHERE code={moduleCode}");
        Result r = db.ResultSet;
        // do something with Result and return module name
    }
}
```

Loose coupling



In this version, we introduce an abstract base class that offers a more neutral interface that could be implemented by different kinds of repository – looser coupling between ModuleCatalogue and Database

Reducing coupling

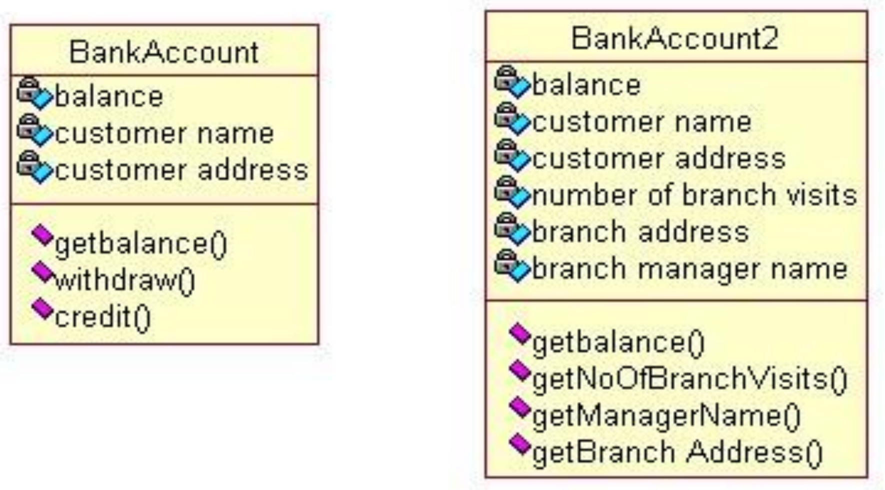
- A **base class/superclass** describes the interface
- All subclasses must implement the interface
- The intention is that if a subclass changes its own specialized interface, it must **still offer** the base class interface in order to continue subclassing the base class
- Using classes – such as ModuleCatalogue – only see the base class and its interface

Code example

```
abstract class Repository {
    abstract Result select(string query);
}

class Database : Repository {
    private string hostname;
    private string dbuser;
    private string dbpassword;
    private string dbname;

    public Result ResultSet { get; }

    void query(string query)
    { /* make query against db and return results */ }

    Result select(string q) {
        this.query(q);
        return this.ResultSet;
    }

    // ... rest of class
}

class ModuleCatalogue {
    private Database db;

    public string lookupModuleName(string moduleCode) {
        Result r = db.select($"SELECT name FROM module WHERE code={moduleCode}");
        // do something with Result and return module name
    }
}
```

Cohesion

- Cohesion is a measure of how logically related are the parts of an individual component
- The greater the cohesion the better, since this means they are more logically related



- Which of these two classes is the most cohesive?

SOLID

- One particular group of important heuristics is known as SOLID
- Acronym of acronyms:
  - **S**RP: Single Responsibility Principle
  - **O**CP: Open-Closed Principle
  - **L**SP: Liskov Substitution Principle
  - **I**SP: Interface Segregation Principle
  - **D**IP: Dependency Inversion Principle
- Principles for object-oriented design (with focus on designing the classes)
- **(This material based on lecture by Yoonsuck Choe)**

SOLID Benefits

Following the SOLID principles helps us design applications and write code with the following benefits:

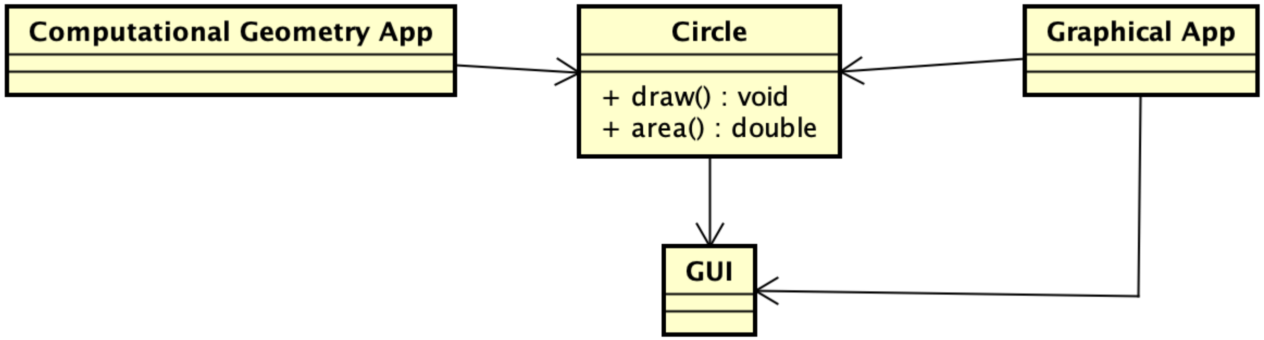
- testable and easily understood
- where things are where they're expected to be
- where classes clearly do what they were intended to do
- can be adjusted and extended quickly without producing bugs
- Separation of public interface from the details (implementation)
- allows for implementations to be swapped out easily

* https://khalilstemmler.com/articles/solid-principles/solid-typescript/

First Pass at Understanding SOLID

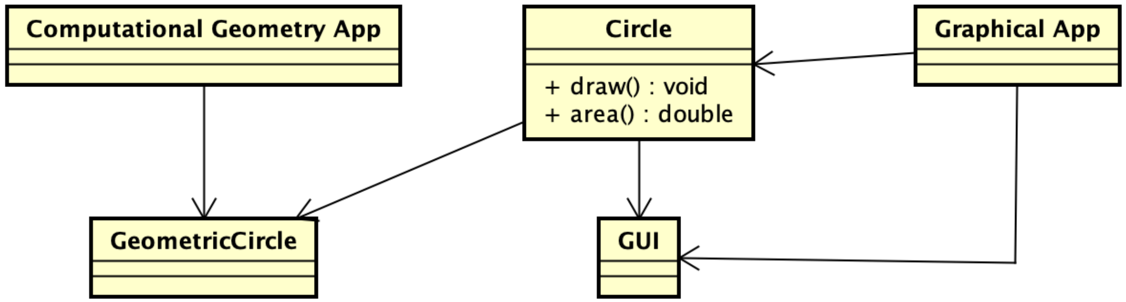
- SRP: “A class should have one, and only one, reason to change”.
- OCP: “You should be able to extend a class’s behavior, without modifying it”
- LSP: “Derived classes must be substitutable for their base classes.”
- ISP: “Make fine grained interfaces that are client specific.”
- DIP: “Depend on abstrations, not on concretions.”

SRP: Single Responsibility Principle



- Example: Circle class with draw() and area()
- Circle combines geometric calculations (area) with graphical operations (draw)
- Computational geometry now depends on GUI, via Circle – even though it doesn't need it
- Any changes to Circle due to Graphical application necessitates rebuild, retest, etc. of Comp. geometry app.
- https://medium.com/@andrewMacmurray/solid-principles-1-single-responsibility-f92ebe986000

SRP: Cont'd



- Solution: Take the purely computational part of the Circle class and create a new class "Geometric Circle".

- All changes regarding graphical display can then be localized into the Circle class.

Open/Closed Principle (OCP)

- Bertrand Meyer (Object-Oriented Software Construction, 1988)
- “software entities (classes, modules, functions) should be open for extension but closed for modification”
- e.g.., we shouldn't change the code in a class, so if we want to modify its behaviour, we need to extend it (through some use of inheritance and/or delegation)
- When requirements change, we extend behaviour, not change old code

Open/Closed - example

- Very simple example

```
class DBConnection {
public:
        void connect(string host,string user,string
            passwd);
};
```

- What if we change the connect method to:

```
bool connect(string host,int port,string user,string
   passwd);
```

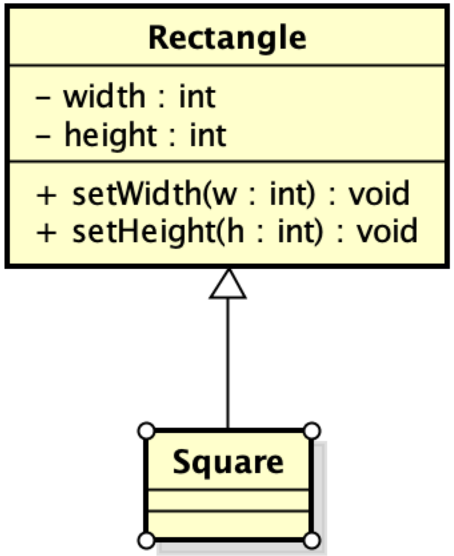
- In this case, the class is not closed, since we have changed existing code – code that calls this method will therefore need to be changed
- We could instead add a new connect method that required the port number, using overloading
  - This extends the class, rather than changing existing code
  - The old method could assume a default port

LSP: Liskov Substitution Principle

- “Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.” (original idea due to Barbara Liskov).
- Violation means the user class’s need to know ALL implementation details of the derived classes of the base class.
- Violation of LSP leads to the violation of OCP.

LSP: Example

- Problem: setWidth(), setHeight() in Rectangle class assumes w and h are independently settable.
- When Square class is used where Rectangle class is called for, behavior can be unpredictable, depending on implementation.
- Want either setWidth() or setHeight() to set both width and height in the Square class.
- LSP is violated when adding a derived class requires modifications of the base class.



LSP: Summary

- LSP is an important property that holds for all programs that conform to the Open-Closed principle.
- If we violate LSP, we violate Open-Closed – if we have to modify the base class, then we are modifying old code
- LSP encourages reuse of base types, and allows **modifications in the derived class** without damaging other components.

ISP: Interface Segregation Principle

- “Clients should not be forced to depend upon interfaces that they do not use.”

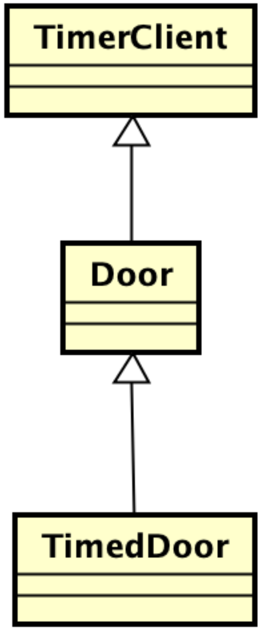
- Avoid “fat interfaces”.

- Fat interfaces: interfaces of a class that can be broken down into groups that serve different set of clients.

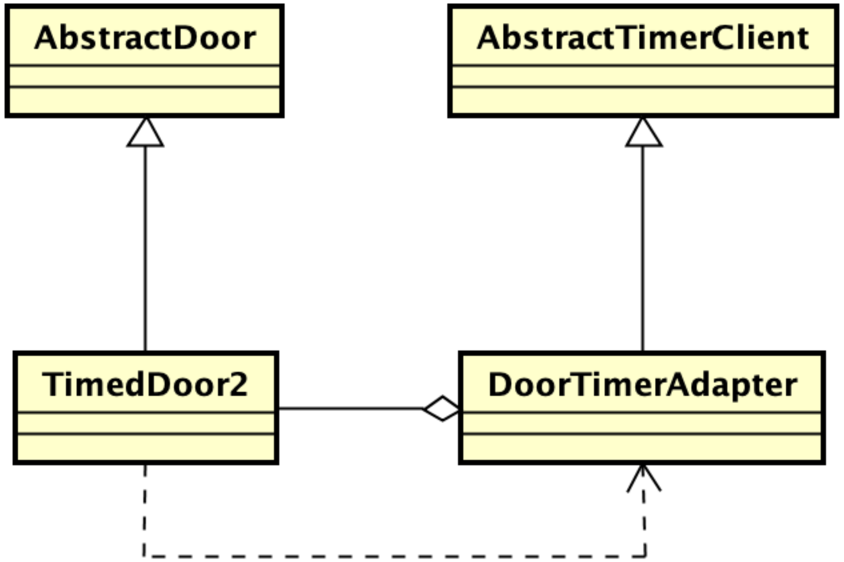
- Clients depending on a subset of interfaces need to change when other clients using a different subset changes.

ISP: Example

- Bad design



- Good design



Clients that use Door or TimerClient access only those speficied interfaces.

ISP Summary

- Example:
  - Bad design: To provide access to time server, Door class inherits from TimerClient unnecessarily.
  - Good design 1: TimedDoor creates DoorTimerAdapter, which includes reference back to TimedDoor, so when triggered, it can send close signal.
  - Good design 2: we could also use multiple inheritance or multiple interfaces
- Should avoid interfaces that are not specific to a single client.
- Fat interfaces cause inadvertant coupling between unrelated clients.

DIP: Dependency Inversion Principle

- “A. High level modules should not depend upon low level modules. Both should depend upon abstractions.”
- “ B. Abstractions should not depend upon details. Details should depend upon abstractions.”
- DIP is an out-growth of OCP and LSP.
- “Inversion”, because standard structured programming approaches make the higher level depend on lower level.

DIP: The Problem

- Consequence of bad design:
  - Hard to change (rigidity)
  - Unexpected parts break when changing code (fragility)
  - Hard to reuse (immobility)

- Causes of bad design:
  - Interdependence of the modules
  - Things can break in areas with NO conceptual relationship to the changed part.
  - Dependent on unnecessary detail.

DIP: Example

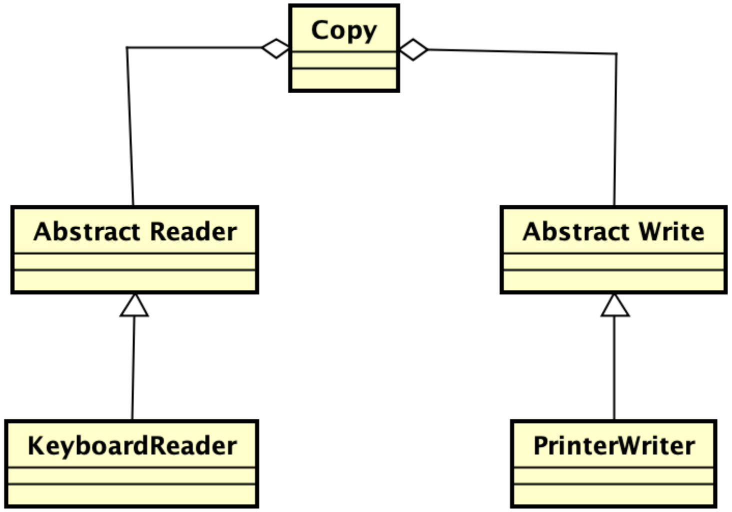
Copy(): uses ReadKeyboard() and WritePrinter(char c);

- Copy() is a general (high-level) functionality we want to reuse.
- The above design is tied to the specific set of hardware, so it cannot be reused to copy over diverse hardware components.
- Also, it needs to take care of all sorts of error conditions in the keyboard and printer component (lots of unncessary details creep in).

DIP: Diagnosis of Copy()

- Module containing high level policy (Copy) is dependent upon low level detailed modules it controls (WritePrinter, ReadKeyboard).

- Good design:

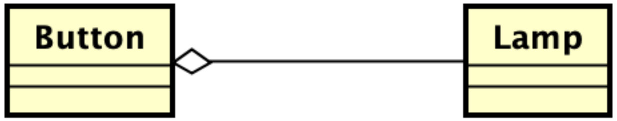


Encourages reuse of abstract interfaces.

DIP: Another Example

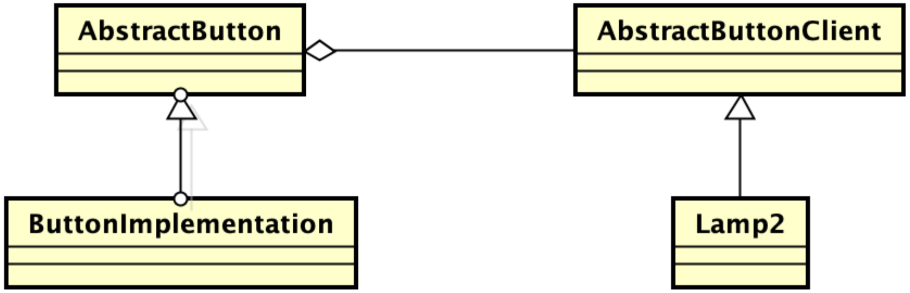
- Bad Design

But



When button changes, lamp has to be at least recompiled. Cannot reuse button for different device.

- Good Design



Button Example – Explained

- Bad design: Button class includes private member Lamp, so that when pressed, it can turn the lamp off. This is bad. When lamp changes, Button is affected.

- Good design: Abstract Button class now containst Abstract Button Client only, so when Lamp changes, Button is not affected.

DIP: Summary

- DIP promises many benefits of OO paradigm.
- Reusability is greately enhanced by DIP.
- Code can be made resilient to change by using DIP.
- As a result, code is easier to maintain.