Object-oriented design principles

Objektumorientált szoftvertervezés
Object-oriented software design

Dr. Simon Balázs
BME, IIT

Outline

- OO concepts
- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP), Design by Contract (DbC)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Release Reuse Equivalency Principle (REP)
- Common Closure Principle (CCP)
- Common Reuse Principle (CRP)
- Acyclic Dependencies Principle (ADP)
- Stable Dependencies Principle (SDP)
- Stable Abstractions Principle (SAP)
- Don’t Repeat Yourself (DRY)
- Single Choice Principle (SCP)
- Law of Demeter (LoD)

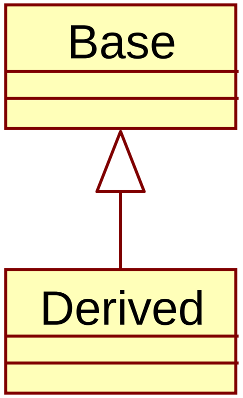
OO concepts

OO concepts

- Class: **type**
  - defines the methods/functions (**behavior**, services) of an object
  - fields/attributes/variables support the behavior and their values define the **state** of an object
- Object: **instance**
  - is an instance of a class
- **Static members**: class members
  - methods and fields corresponding to the class as a whole
  - only **one copy** across all objects
  - static methods have no this pointer, cannot access instance members directly
- **Instance members**: object members
  - methods and fields corresponding to an object
  - **different copy** for each object
  - instance methods have a common implementation, but they receive an implicit 0^{th} parameter: the **this** pointer (current object instance)

OO concepts

- **Abstraction**:
  - ignoring the unnecessary details in the given context
  - real-world objects can be mapped to objects in a program
- **Classification**:
  - grouping things with common behavior and properties together
  - objects with common behavior and properties can be described by the same class
- **Encapsulation**:
  - a class should not allow direct access to the fields
  - only through methods
  - fields should be private



- **Inheritance**:
  - a derived class of a base class can reuse the base class's behavior
  - inheritance means "is-a-type-of" relationship between the derived class and the base class
  - important: never use inheritance for data reuse! use delegation instead!
- **Polymorphism**:
  - the caller of a method does not have to worry whether an object belongs to a parent class or one of its descendants
  - realized by virtual methods and inheritance: virtual methods can be overridden in derived classes

OO concepts

- Visibility:
  - **private** (-): can only be accessed by the current class
  - **protected** (#): can only be accessed by the current class or by a derived class
  - **public** (+): can be accessed by anyone who has access to the class
  - **package** (~): can be accessed from the same package
- **Virtual** method:
  - virtual methods can be **overridden** in a derived class
  - this is a way to **extend the behavior** of a base class
- **Abstract method**:
  - a virtual method with no implementation
- **Abstract class**:
  - abstract classes **cannot be instantiated**
  - usually it has at least one abstract function, but this is not necessary
- **Interface**:
  - a set of operations with no implementation
  - defines the expected behavior/contract of a class implementing the interface
- **Interface of a class**:
  - the set of public methods of a class

OO concepts

- Coupling:
  - manner and degree of interdependence between software modules/classes/functions
  - measure of how closely connected two routines or modules are
  - the strength of the relationships between modules
  - **low coupling** is good for maintainability: a change in one part of the system implies only a few changes in other parts of the system
- Cohesion:
  - degree to which the elements of a module/class belong together
  - measures the strength of relationship between pieces of functionality within a given module
  - **high cohesion** is good for maintainability: in highly cohesive systems functionality is strongly related, therefore, changing the functionality is localized

OO design principles

Problem of change

- Software is subject to change
- A good design can make change less troublesome
- The problem is when the requirements change in a way that was not anticipated by design
- If the design fails under changing requirements, then it is the design’s fault
- Changes in a later phase may violate the original design philosophy, and these changes can escalate
  - this is why documentation is important

Bad design

- The design is bad when:
  - it is hard to change because every change affects too many other parts of the system (Rigidity)
  - changes break down unexpected parts of the system (Fragility)
  - it is hard to reuse parts of the system in another application because these parts cannot be disentangled from the current application (Immobility)
- Cause of bad design: heavy interdependence between the parts of the application
- Solution:
  - reduce dependency between parts
  - drive dependency away from problematic or frequently changing parts

SOLID principles

- Five principles of good OO design
  - Single responsibility
  - Open-closed
  - Liskov substitution
  - Interface segregation
  - Dependency inversion
- Introduced by Robert C. Martin
- These principles promote maintainability and extensibility over time by reducing dependency between parts of an application

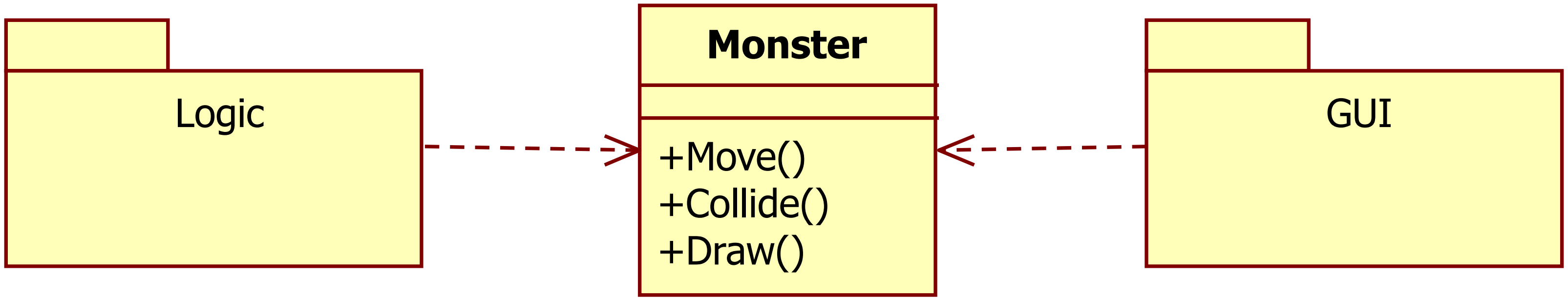
Single Responsibility Principle (SRP)

- A class should have only one reason to change
  - (Robert C. Martin)
- In this context: responsibility = reason to change (≠ the provided services)
- This means that if a class has more than one responsibilities it should be split into multiple classes

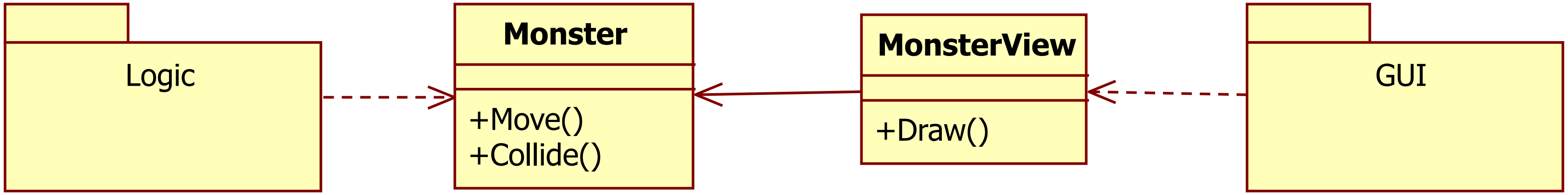
SRP

- If a class has more than one responsibilities, split the responsibilities:
  - at implementation level (if they can be uncoupled)
  - at interface level (if they cannot be uncoupled)
- Implementation level:
  - separate classes
  - one using the other (e.g. GUI using business logic) but not vice versa
- Interface level:
  - interfaces for separate responsibilities
  - implement both interfaces in the same class
- Advantage: dependencies flow away from the problematic responsibilities

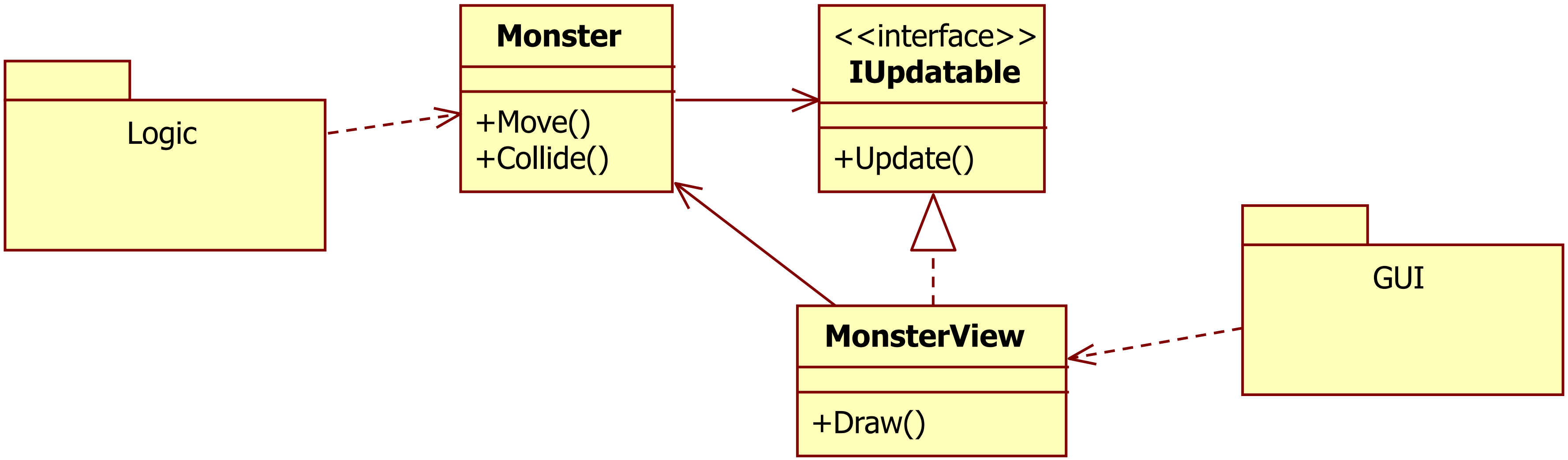
SRP violation example



SRP solution I.



SRP solution II.



Probability of change

- It is not always obvious that there are more reasons to change
- It is also possible that the change will never occur
- We have to estimate the probability of the change based on our past experiences and on our domain knowledge
- Don’t design for change if it has low probability
  - YAGNI = You Ain’t Gonna Need It

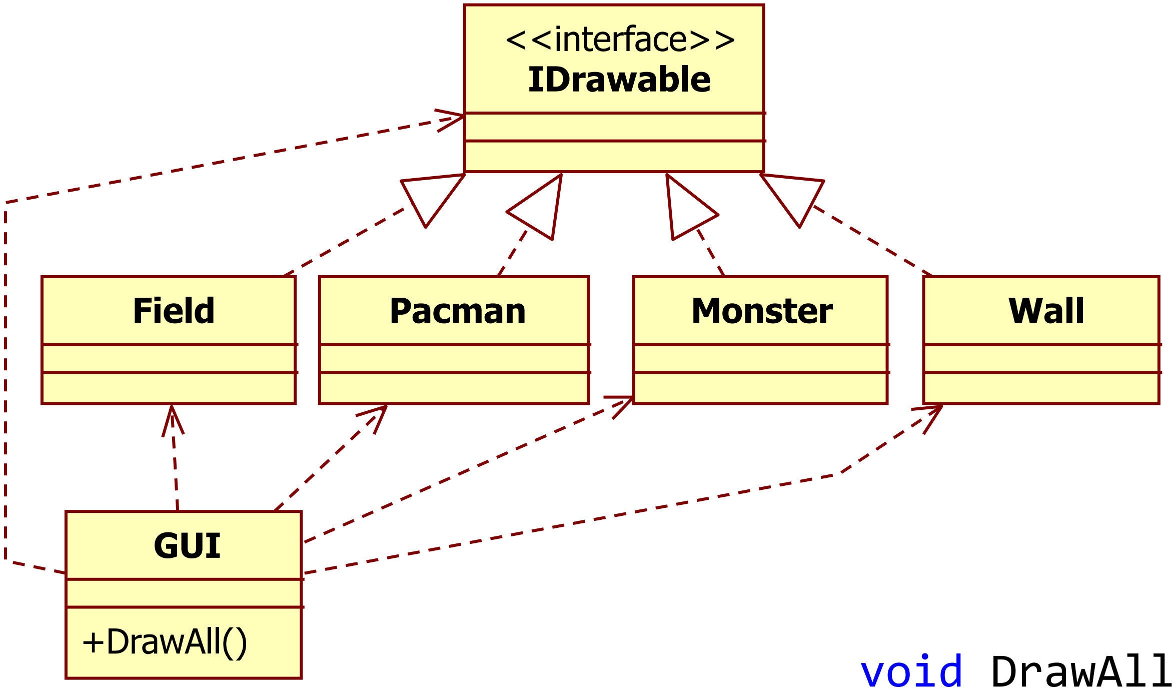
Open/Closed Principle (OCP)

- Software entities (classes, modules, functions etc.) should be open for extension, but closed for modification.
  - (Bertrand Meyer)
- Open for extension: the behavior of the module can be extended to satisfy the changes in the requirements
- Closed for modification: extending the behavior of the module does not result in changes to the source of the module

OCP

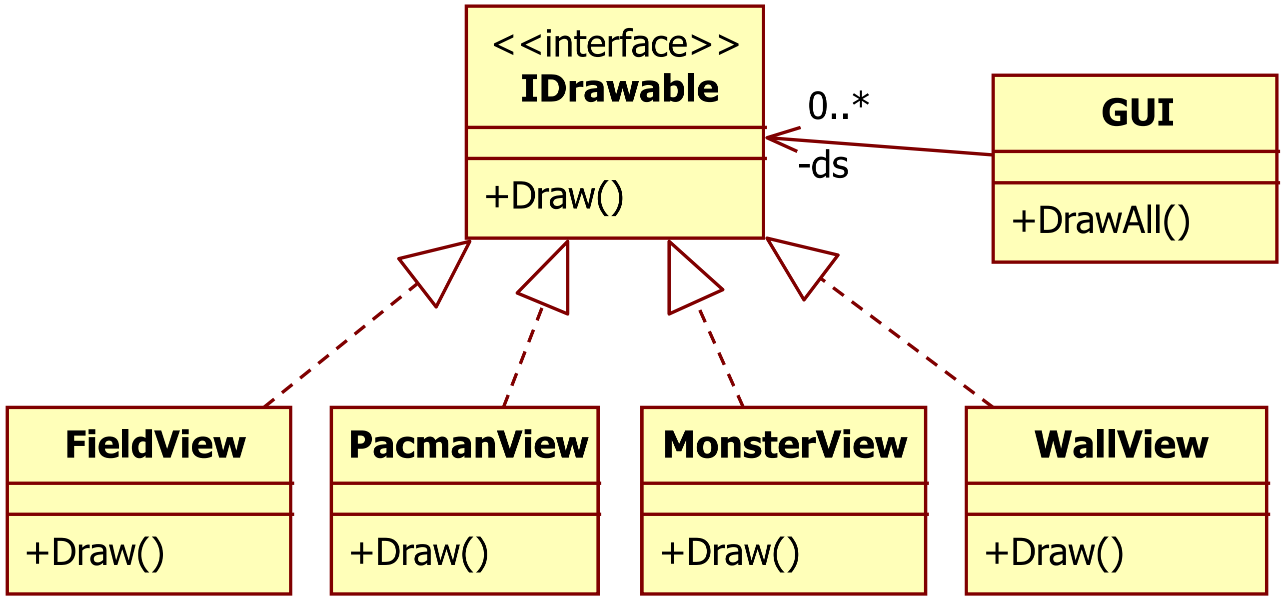
- Prepare for change
- Open for extension:
  - new subclasses
  - overriding methods
  - polymorphism
  - delegation
- Closed for modification:
  - original code does not change
  - only bugfixes
- Extend behavior by adding new code, not changing existing code

OCP violation example




```
void DrawAll(List<IDrawable> ds) {
  foreach (var d in ds) {
    if (d is Field) {
      // ...draw field...
    } else if (d is Pacman) {
      // ...draw pacman...
    } else if ...
  }
}
```

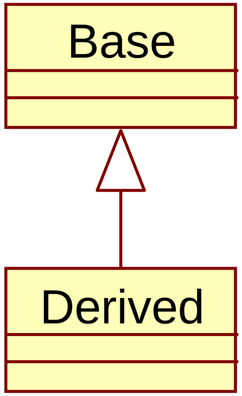
OCP solution




```
void DrawAll(List<IDrawable> ds) {
  foreach (var d in ds) {
    d.Draw();
  }
}
```

Liskov Substitution Principle (LSP)

- Subtypes must be substitutable for their base types
  - (Barbara Liskov)
- Any Derived class object must be substitutable where ever a Base class object is used, without the need for the user to know the difference
- Inheritance:
  - Derived is a kind of Base
  - every object of type Derived is also of type Base
  - what is true for an object of Base it is also true for an object of Derived
  - Base represents a more general concept than Derived
  - Derived represents a more specialized concept than Base
  - Anywhere an object of Base can be used, an object of Derived can be used



LSP

- The importance of this principle becomes obvious when the consequences of its violations are considered
- Violations of LSP:
  - the derived behaves differently than it is expected from the base
  - typically:
    - inheritance for data reuse (e.g. square-rectangle problem)
    - exceeding the range of input/output parameters
- Consequences of this:
  - explicit type check is required to recognize the misbehaving subclass
    - is, instanceof, dynamic_cast, etc.
  - hence: violation of OCP

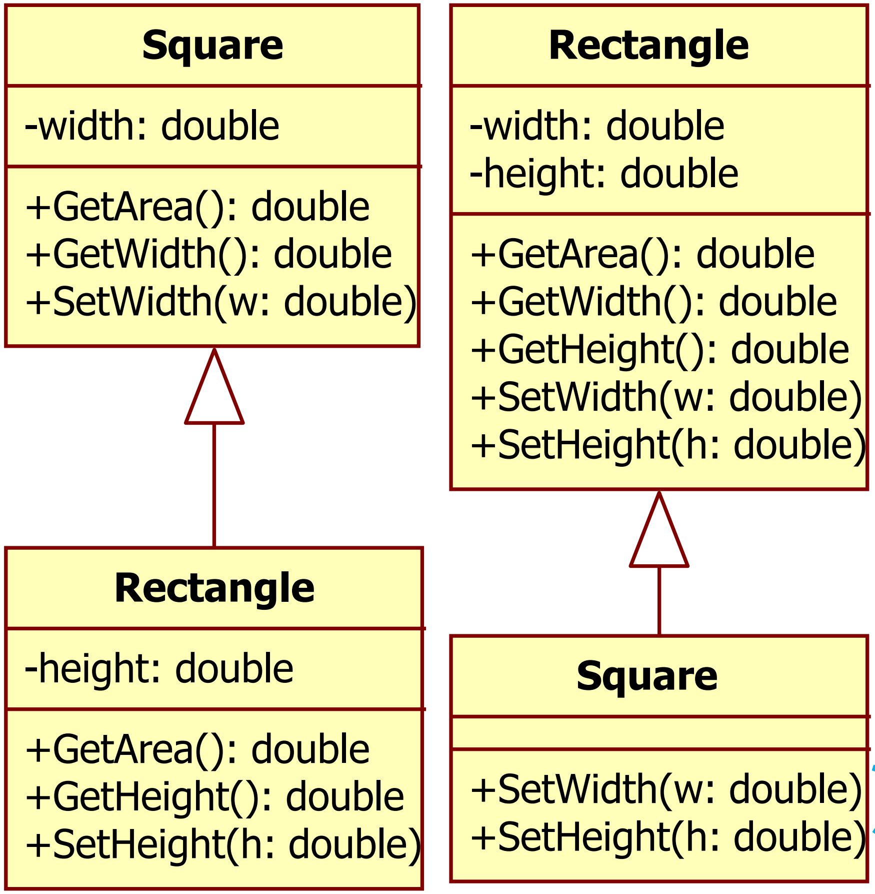
LSP violation: type check

- If a descendant class violates LSP, an inexperienced developer or a developer in a hurry may fix the problem with explicit type check for the problematic type
- For example:

```
void Draw(Shape s) {
  if (s is Rectangle) DrawRectangle((Rectangle)s);
  else if (s is Ellipse) DrawEllipse((Ellipse)s);
}
```

  - Here Draw violates OCP, since it must know about every possible derivative of Shape
  - Violation of LSP also leads to the violation of OCP

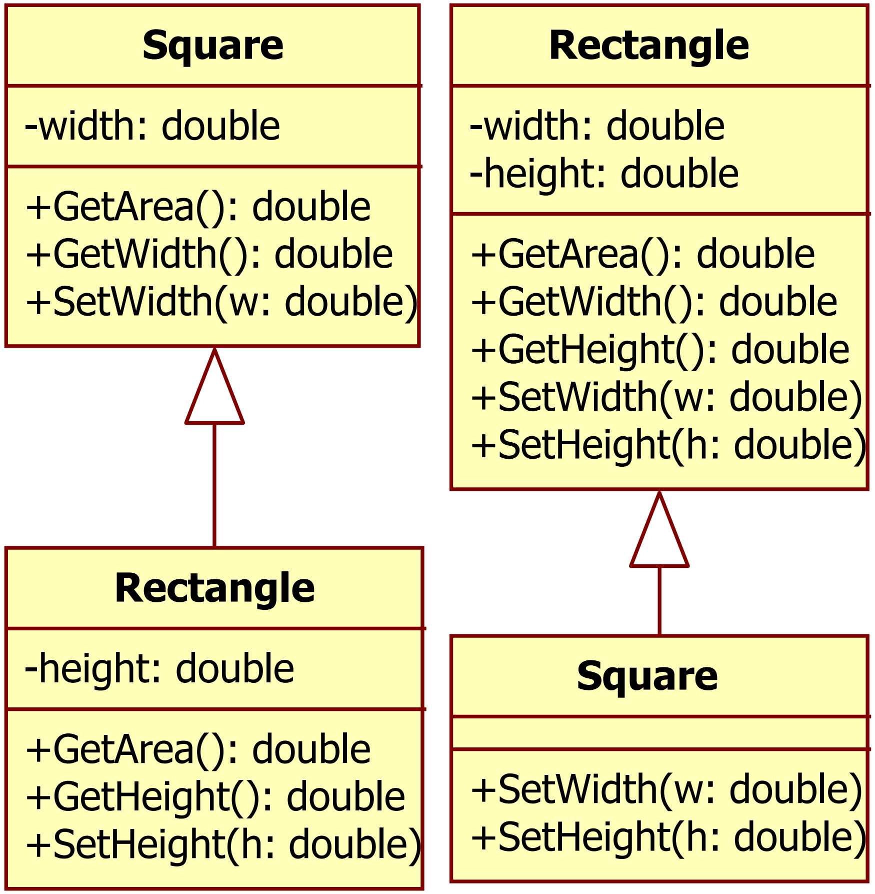
Which design is better?



Neither: both of them violate LSP

```
class Square : Rectangle {
  void SetWidth(double w) {
    base.SetWidth(w);
    base.SetHeight(w);
  }
  void SetHeight(double h) {
    base.SetWidth(h);
    base.SetHeight(h);
  }
  // ...
}
```

Both violate LSP




```
void TestSquare(Square s) {
  s.SetWidth(5);
  Debug.Assert(s.GetArea() == 25);
}

void TestRectangle(Rectangle r) {
  r.SetWidth(5);
  r.SetHeight(4);
  Debug.Assert(r.GetArea() == 20);
}
```

LSP

- The root of the square-rectangle problem:
  - square is a kind of rectangle from the mathematic point of view (data)
  - but square has a different behavior than a rectangle (SetWidth should not change the height)
  - and the behavior is what counts in OO
- Another important warning: never use inheritance for data reuse, always inherit for behavior reuse

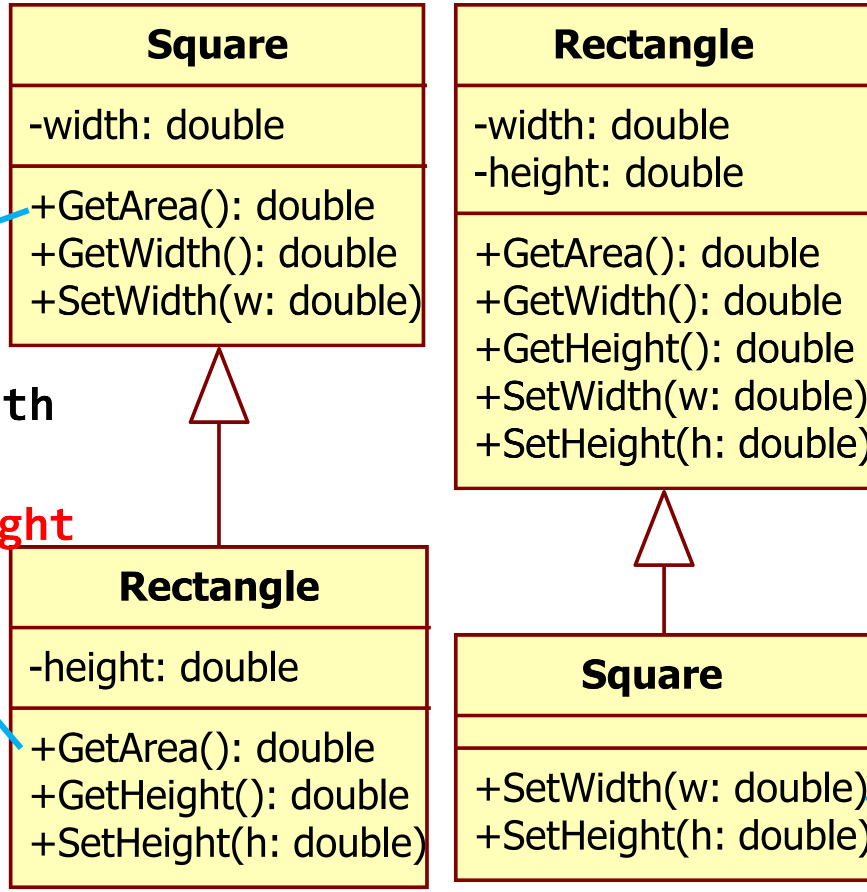
Pre- and post-conditions, invariants

- Pre-conditions of a method:
  - conditions which must be true before the method is called
  - pre-conditions must be fulfilled by the client (caller)
  - examples:
    - Stack.Pop(): stack must not be empty
    - Math.Sqrt(x): x must be non-negative
  - if a pre-condition is not met, the method usually throws an exception
- Post-conditions of a method
  - conditions which are true after the method is executed
  - post-conditions must be fulfilled by the server (called object)
  - examples:
    - Stack.Push(item): the topmost element of the stack is item
    - Math.Sqrt(x): the result is non-negative
- Class invariants:
  - constraints which are true about the state of an instance of the class (object)
  - established during construction and maintained between public method calls
  - examples:
    - Hour field of a Time object: a number between 0 and 23
    - Sides of a triangle: sum of any two sides exceeds the third side

Design-by-Contract

- Advertised behavior of the Derived class is substitutable for that of the Base class *(Bertrand Meyer)*
- Contract is advertised behavior:
  - set of pre-conditions, post-conditions and invariants
- The rule for inheritance/substitutability:
  - A derived method may only replace the original pre-condition by one equal or weaker, and the original post-condition by one equal or stronger. Invariants can be strengthened but not weakened.
  - *Expect no more, provide no less. / Demand no more, promise no less.*
- For example, the post-condition of SetWidth in the Rectangle class would be:
  - assert(width == w && height == old.height)
  - that is, the height remains unchanged
  - this is violated by the Square class

Violation of post-conditions



GetArea post:
`result=width*width`

GetArea post:
`result=width*height`

SetWidth post:
`width == w &&`
`height == old.height`

SetHeight post:
`height == h &&`
`width == old.width`

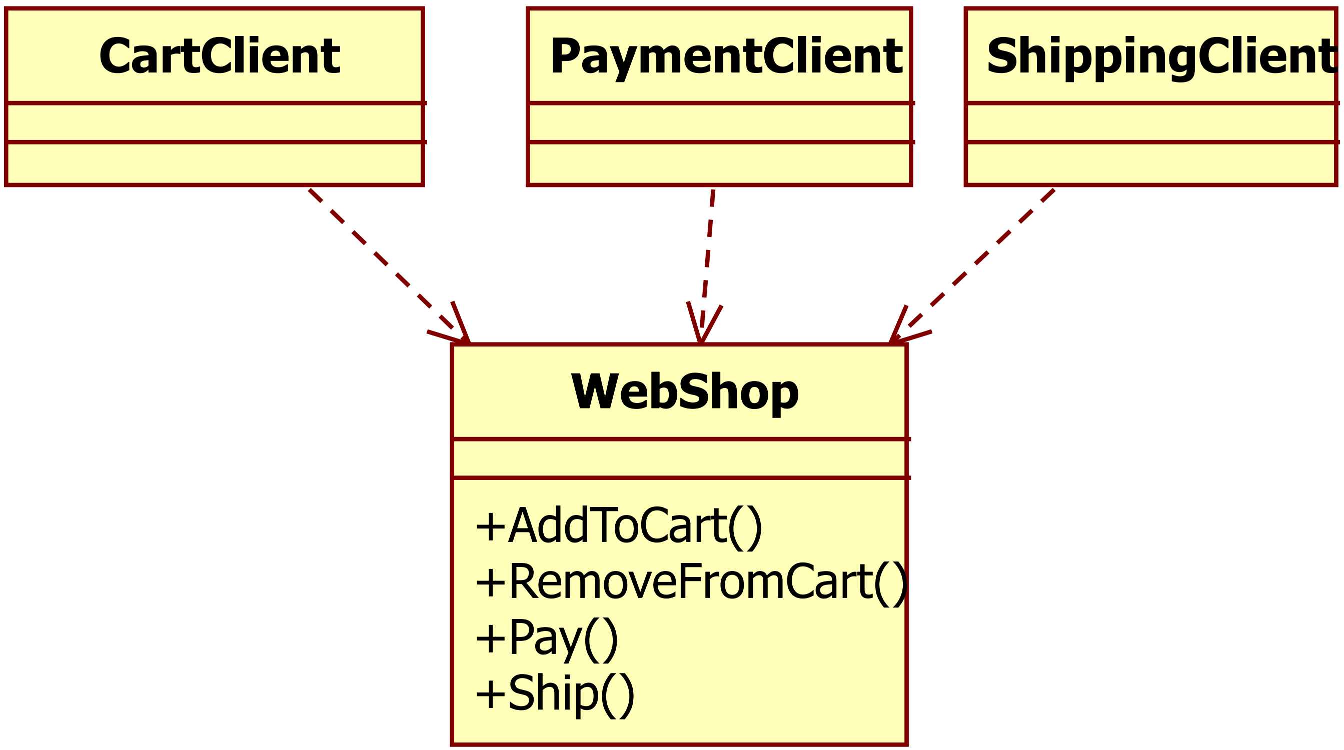
SetWidth post:
`width == w &&`
`height == w`

SetHeight post:
`height == h &&`
`width == h`

Interface Segregation Principle (ISP)

- Clients should not be forced to depend on methods they do not use
  - (Robert C. Martin)
- ISP acknowledges that objects require fat, non-cohesive interfaces
- However, clients should not know about them as a single class
- Instead, clients should know about abstractions with cohesive interfaces

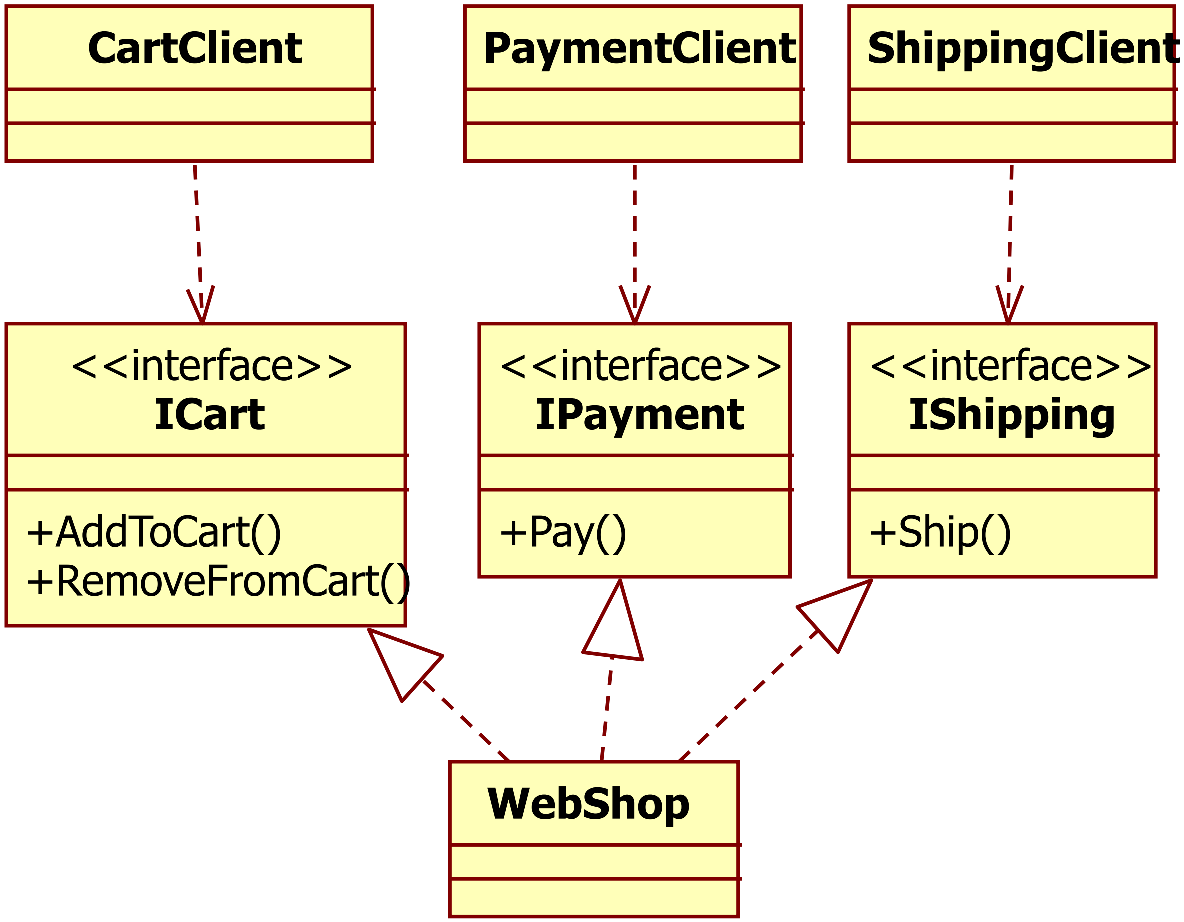
ISP violation example



ISP

- The clients should depend on the methods they actually call
- Break up the polluted or fat interface of a class into many client-specific interfaces
- Let the class implement all these interfaces
- The clients should depend only on the interface they require
- This breaks the dependence of clients on methods they do not use
- And hence, the clients will be independent of each other

ISP solution

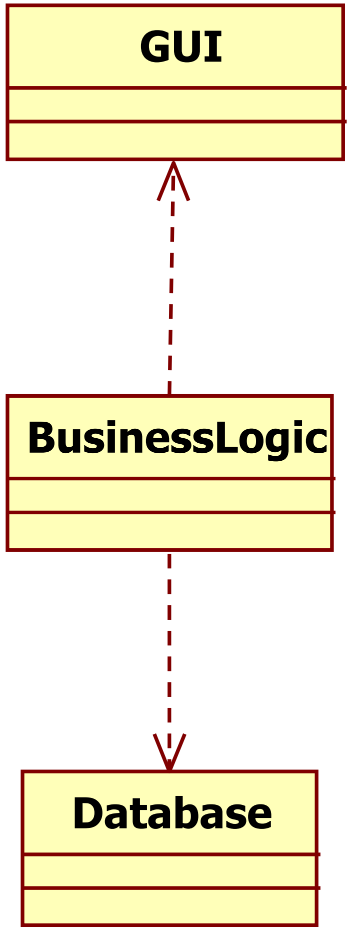


Dependency Inversion Principle (DIP)

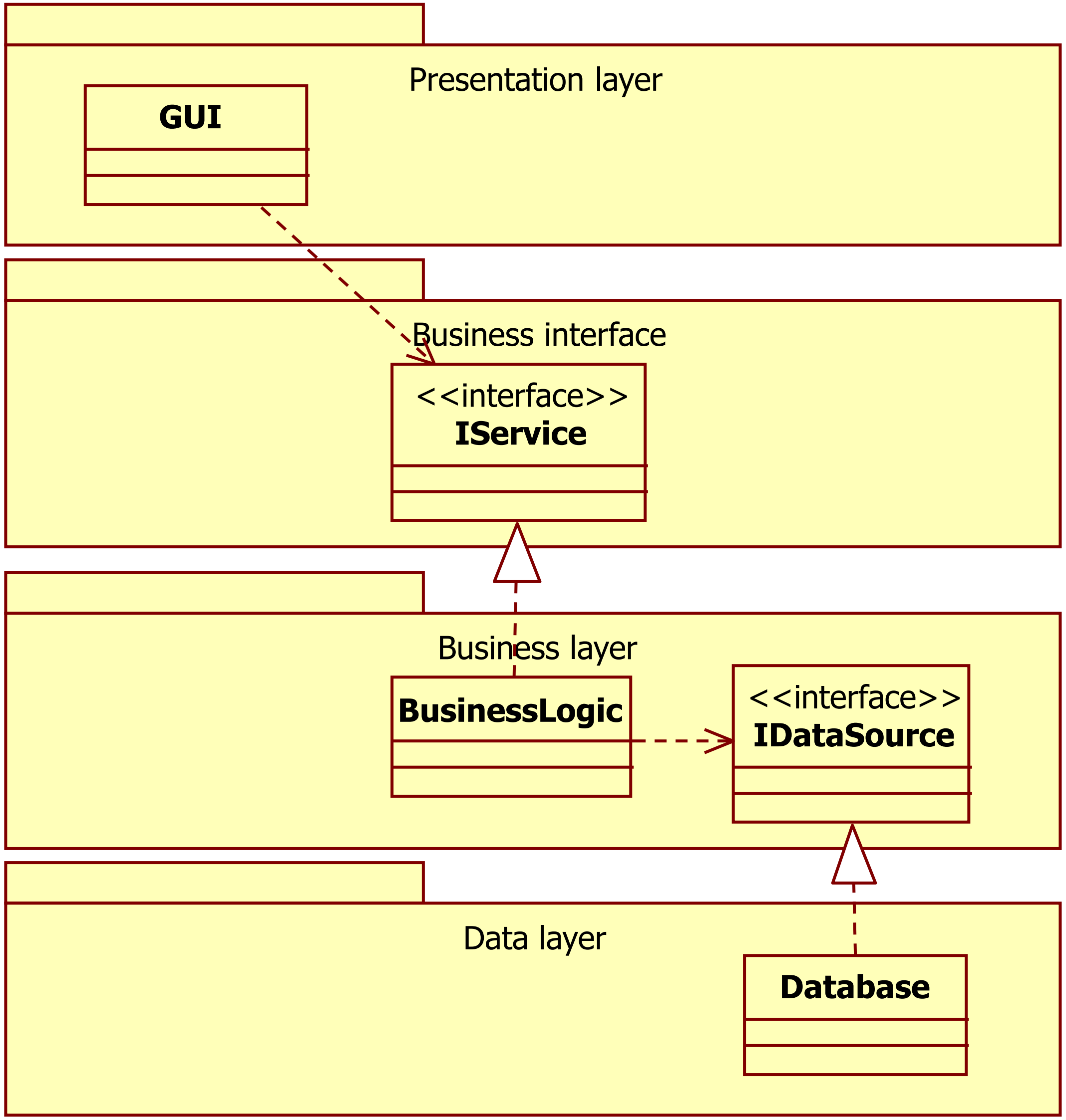
- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.
  - (Robert C. Martin)
- An application consists of modules
- A natural way to implement these modules is to write the low-level modules (input-output, networking, database, etc.) and combine them in high-level modules
- However, this is bad: a change in a low-level module affects high-level modules

DIP violation example

- Example for bad design:
  - Business logic ---> Database
  - Business logic ---> GUI
  - the concrete database technology or
    the concrete GUI technology can change
  - changing the database or the GUI
    induces changes in the business logic
- Dependency Inversion Principle:
  - change the direction of the dependency:
    let low-level modules depend on
    abstractions defined by high-level modules



DIP solution



DIP

- Another interpretation of DIP: depend on abstractions
- According to this in the strong sense:
  - no instances of a concrete class can be created
  - no classes should be derived from a concrete class
  - no method should override an implemented method of any of its base classes
- Clearly, these are too strong conditions, and the first one is always violated
- Use creational patterns to inject concrete instances for abstractions (e.g. abstract factory)
- If a concrete class is not going to change very much, and no other similar derivatives are going to be created, then its perfectly OK to depend on it
  - e.g. String class in C#/Java

DIP criticism

- There are times when the interface of a concrete class has to change
- And this change must be propagated to the abstract interface that represents the class
- A change like this will break through of the isolation of the abstract interface
- However, the client classes declare the service interfaces they need, and the only time this interface will change is when the client needs the change

Release Reuse Equivalency Principle (REP)

- A reusable element (module, component, class, etc.) cannot be reused unless it is managed by a release system
- The released elements must have version numbers
- The author of the reusable element must be willing to support and maintain older versions until the users can upgrade to newer versions
- Otherwise, the released elements won't be reused
- Since packages are the unit of release, reusable classes should be grouped into packages

Common Closure Principle (CCP)

- Classes that change together, belong together
- The more packages that change in a release, the greater the work is to adjust, rebuild, test and deploy the new release
- Thus, classes that are likely to change together should be grouped into the same package
- We have to anticipate the possible changes, and decide based on these

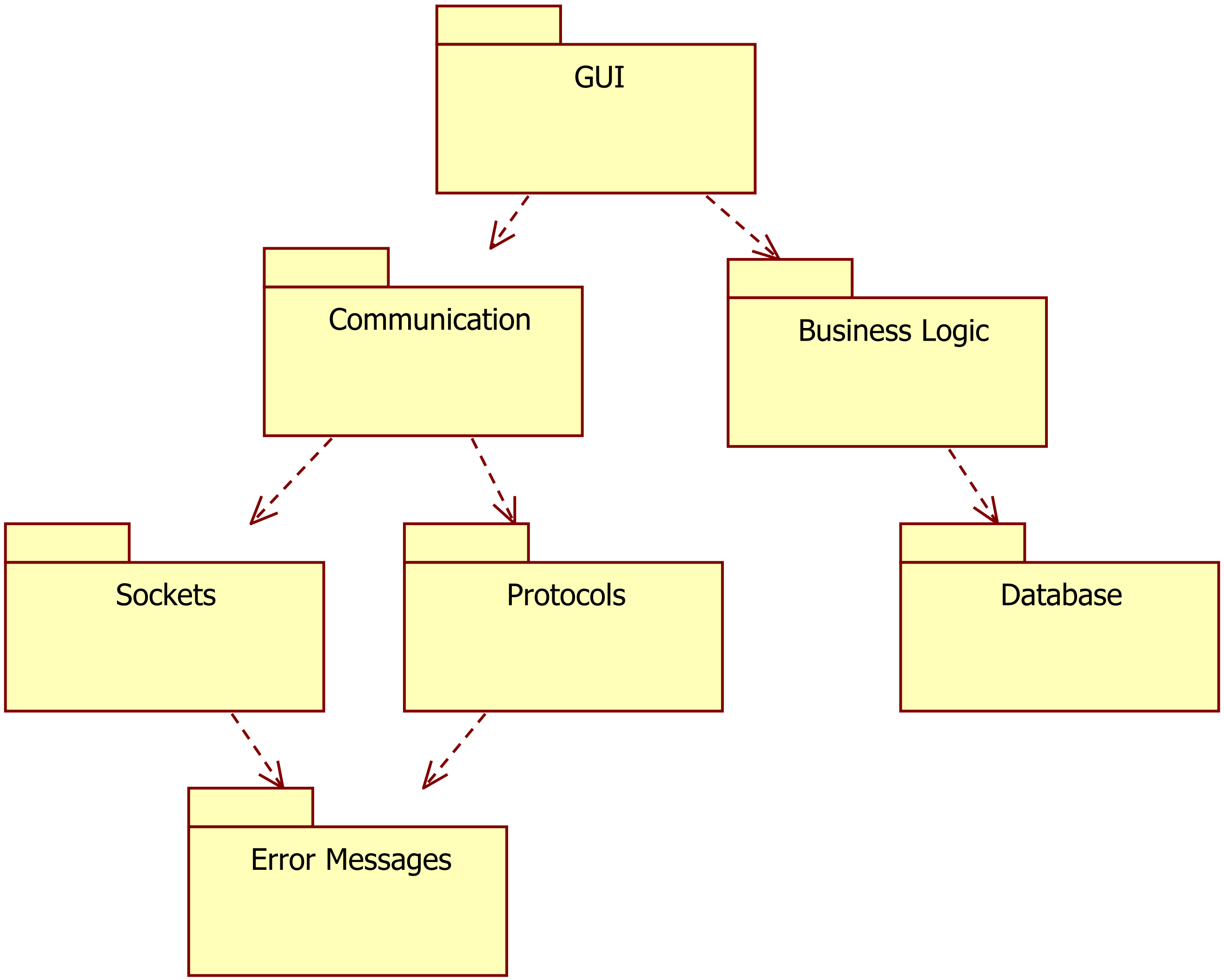
Common Reuse Principle (CRP)

- Classes that aren't reused together should not be grouped together
- Dependency upon a package is a dependency upon everything in the package
- When a package changes, all its users have to be changed, even if they do not use the changed elements from the package
- This is similar to ISP at package level

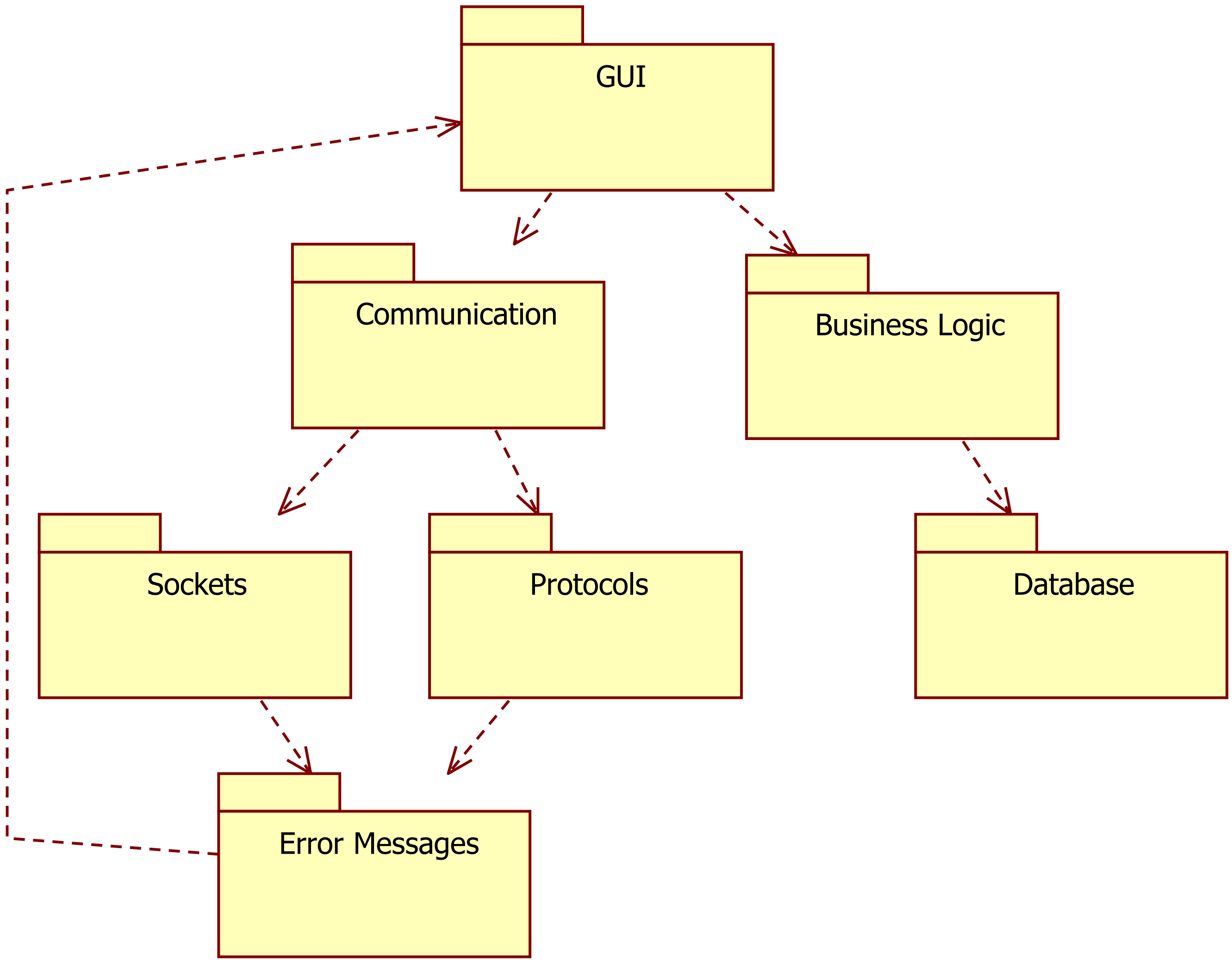
Acyclic Dependencies Principle (ADP)

- Dependencies between packages must not form cycles
- A newly released package must be tested with its dependencies
- Hopefully, the number of dependencies is small
- But if there is a cycle in the dependency graph, all the packages in the cycle must be rebuilt and retested

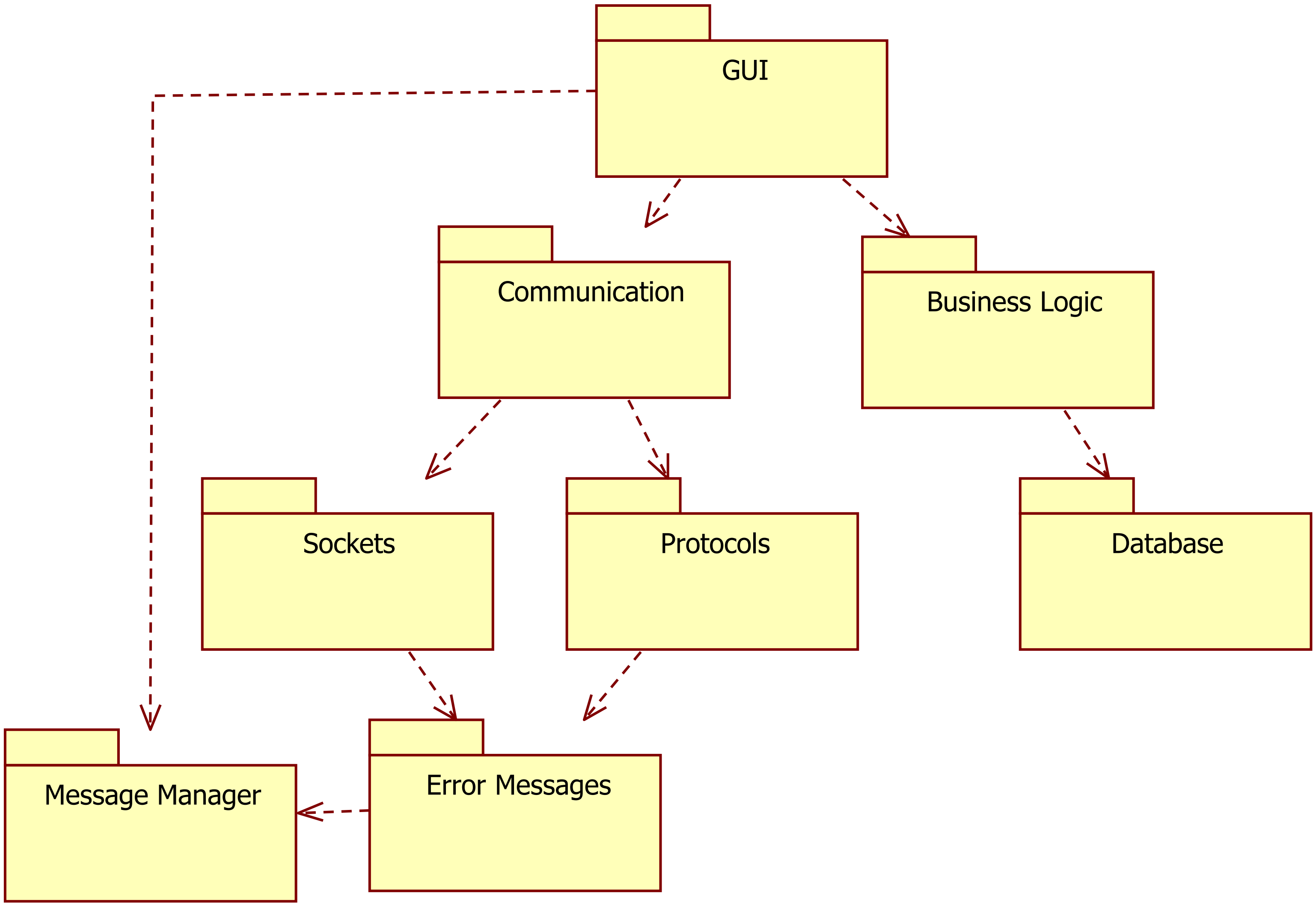
ADP


GUI
Communication
Business Logic
Sockets
Protocols
Database
Error Messages

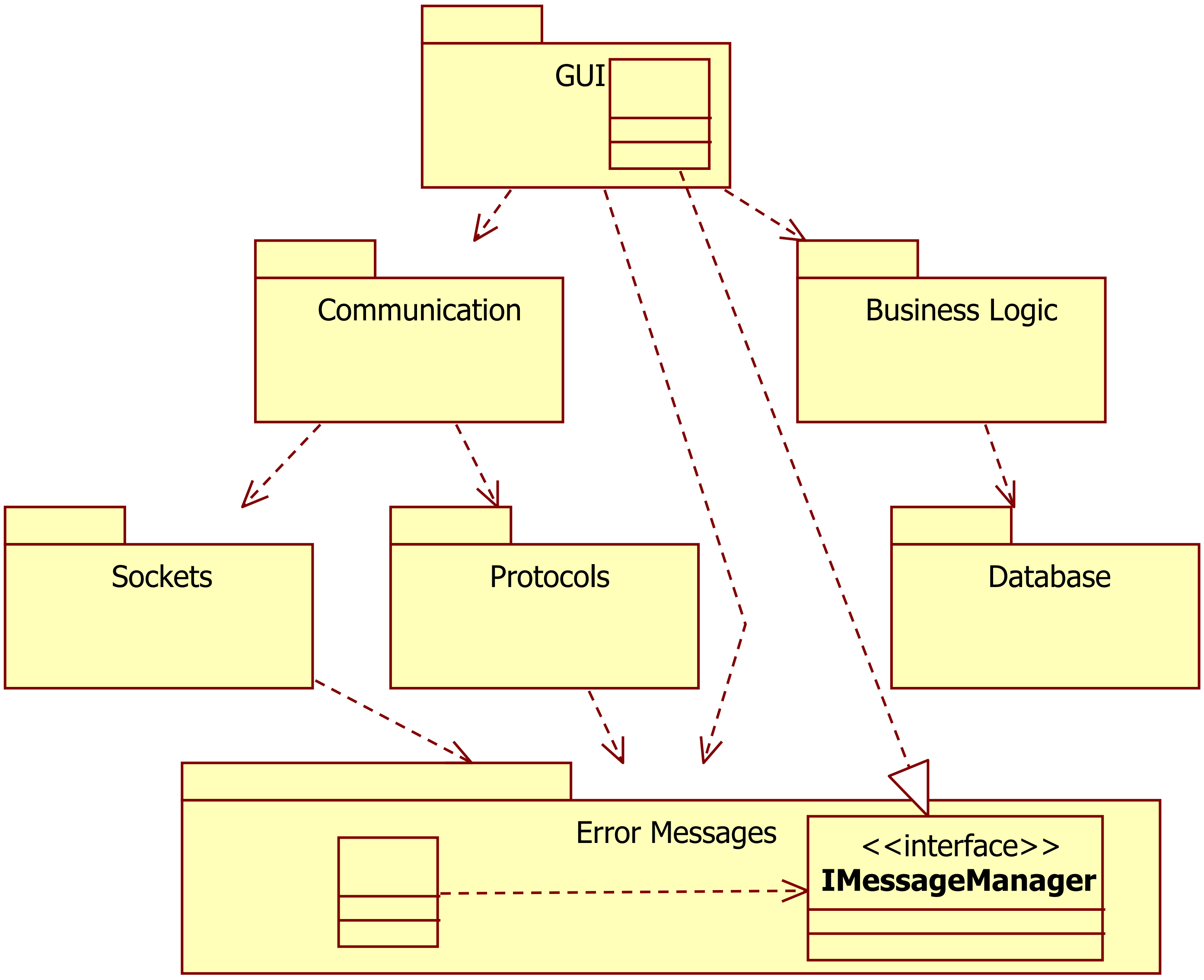
ADP violation example



ADP solution I: introducing a new package

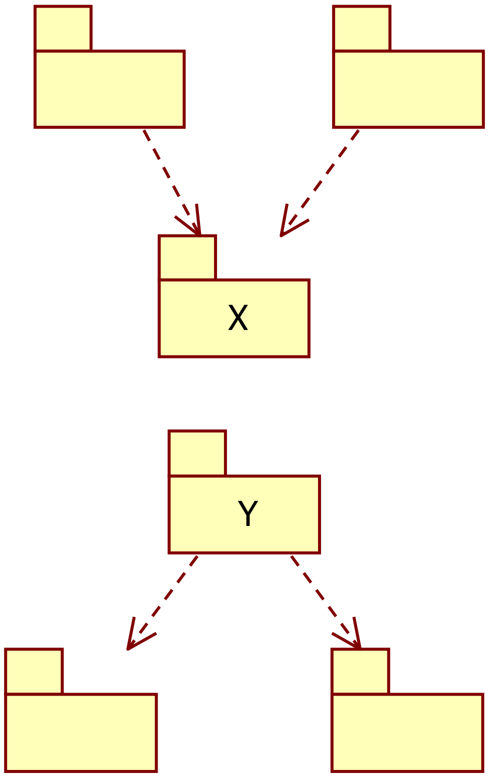


ADP solution II: DIP and ISP



Stable Dependencies Principle (SDP)

- Depend in the direction of stability
- Stability is related to the amount of work to make a change
- X is stable:
  - two packages depend on it, this is good reason not to change X; X is responsible to these packages
  - X depends on nothing; X is independent
- Y is instable:
  - Y has no other packages depending on it; Y is irresponsible
  - Y depends on two other packages, so change can come from two directions; Y is dependent



SDP

- Software is subject to change
- A good design means that changes can be made easily
- So the most frequently changing parts of the application should be instable, since changes in an instable package cost less
- Stable packages must never be dependent on flexible and continuously changing packages
- Examples:
  - the GUI should depend on the model
  - the model should not depend on the GUI

Stable Abstractions Principle (SAP)

- Stable packages should be abstract packages
- The software is built of packages:
  - instable and flexible packages at the top
  - stable and difficult to change packages at the bottom
- Question: is stability good?
  - Although stable packages are difficult to change, they do not have be difficult to extend!

SAP

- If stable packages are highly abstract, they can be easily extended
- So the software is built of packages:
  - instable and flexible packages at the top that are easy to change
  - stable and highly abstract packages at the bottom that are easy to extend
- SAP is just another form of DIP

Don’t Repeat Yourself (DRY)

- Every piece of knowledge must have a single, unambiguous, authoritative representation within a system
- Reduce repetition
- Duplication is bad:
  - if something has to be changed in the repetitive part of the code, it has to be changed in all occurrences
  - there is a high probability that some occurrences will be missed
- If you hit Ctrl+C think about creating a method from the copied part of the code

DRY violation example

```
class Producer {
  private Queue queue;
  public void Produce() {
    lock (queue) {
      string item = "hello";
      queue.Enqueue(item);
    }
  }
}

class Consumer {
  private Queue queue;
  public void Consume() {
    lock (queue) {
      string item = queue.Dequeue();
    }
  }
}
```

DRY solution: making the Queue thead-safe

```
class Queue {
  private object mutex = new object();
  private List<string> items = new List<string>();

  public void Enqueue(string item) {
    lock (mutex) {
      items.Add(item);
    }
  }

  public string Dequeue() {
    lock (mutex) {
      string result = items[0];
      items.RemoveAt(0);
      return result;
    }
  }
}
```

DRY solution: making the Queue thead-safe

```
class Producer {
  private Queue queue;
  public void Produce() {
    string item = "hello";
    queue.Enqueue(item);
  }
}

class Consumer {
  private Queue queue;
  public void Consume() {
    string item = queue.Dequeue();
  }
}
```

Single Choice Principle (SCP)

- Whenever a software system must support a set of alternatives, ideally only one class in the system knows the entire set of alternatives
- This is a corollary to DRY and OCP
- Also known as: Single Point Control (SPC)
  - The exhaustive list of alternatives should live in exactly one place

Tell, don’t ask (TDA)

- Methods should be called without checking the target object’s state or type at first
- The state check is a behavior that should belong to the target object’s responsibilities

TDA violation example

```
class Pacman {
  void Step() {
    Field next = field.GetNext();
    if (next.IsFree) {
      next.Accept(this);
    } else {
      Thing other = next.GetThing();
      other.Collide(this);
    }
  }
}
```

TDA solution

```
class Pacman {
  void Step() {
    Field next = field.GetNext();
    field.Remove(this);
    next.Accept(this);
  }
}

class Field {
  void Accept(Thing t) {
    if (this.thing != null) {
      this.thing.Collide(t);
    } else {
      this.thing = t;
    }
  }
}
```

TDA

- Violation of TDA also violates DRY
- It can cause a lot of problems:
  - checking the condition may be forgotten in some places
  - concurrency problems
  - pre-condition violation problems
- Leave the checking of conditions to the object being called
- The violation of TDA means that the responsibilities are not at the right place

Law of Demeter (LoD)

- “Don’t talk to strangers!”
- A method of an object should only invoke methods of:
  - itself
  - its parameters
  - its members
  - objects it creates
- A method should not invoke any other objects’ members

LoD violation example

Either:

```
this.field.GetNext().GetThing().Collide(this);
```

Or:

```
Field next = this.field.GetNext();
Thing thing = next.GetThing();
thing.Collide(this);
```

LoD possible solutions

```
//Acceptable:
Field next = this.field.GetNext();
next.Accept(this);


//Better:
this.field.MoveThingToNextField();
```

LoD

- Chaining method calls means dependency on all the items of the chain
- Solution: provide a method in each object to delegate the call to the next object
- But make sure there is no combinatorial explosion of methods
  - if there is, redesign responsibilities
- Only delegate to own members!
- This way if an item in the chain changes it probably won't affect the object at the beginning

Summary

- Single Responsibility Principle (SRP)
- Open/Closed Principle (OCP)
- Liskov Substitution Principle (LSP), Design by Contract (DbC)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)
- Release Reuse Equivalency Principle (REP)
- Common Closure Principle (CCP)
- Common Reuse Principle (CRP)
- Acyclic Dependencies Principle (ADP)
- Stable Dependencies Principle (SDP)
- Stable Abstractions Principle (SAP)
- Don’t Repeat Yourself (DRY)
- Single Choice Principle (SCP)
- Law of Demeter (LoD)