

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 0th 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:

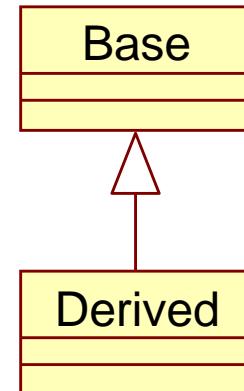
- grouping things with common behavior and properties together
- objects with common behavior and properties can be described by the same class

■ 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

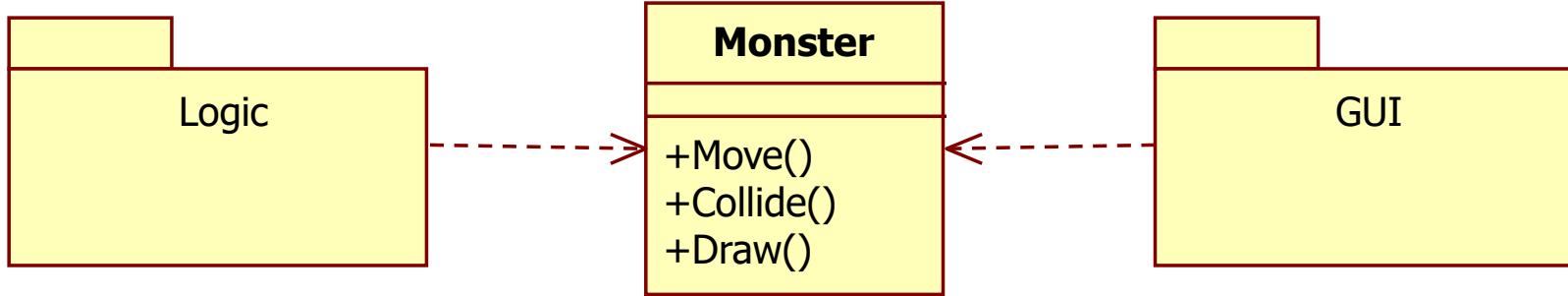
- Five principles of good OO design
 - Single responsibility
 - Open-closed
 - Liskov substitution
 - Interface segregation
 - Dependency inversion
- Introduced by Robert C. Martin
- These principles promote maintainability and extensibility over time by reducing dependency between parts of an application

Single Responsibility Principle (SRP)

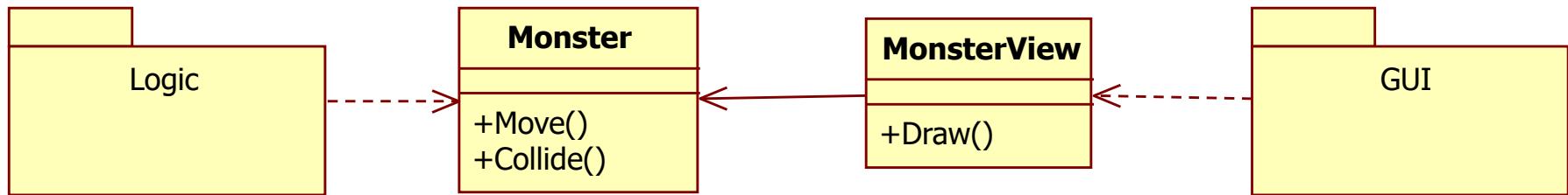
- A class should have only one reason to change
 - (Robert C. Martin)
- In this context: responsibility = reason to change (\neq the provided services)
- This means that if a class has more than one responsibilities it should be split into multiple classes

- 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

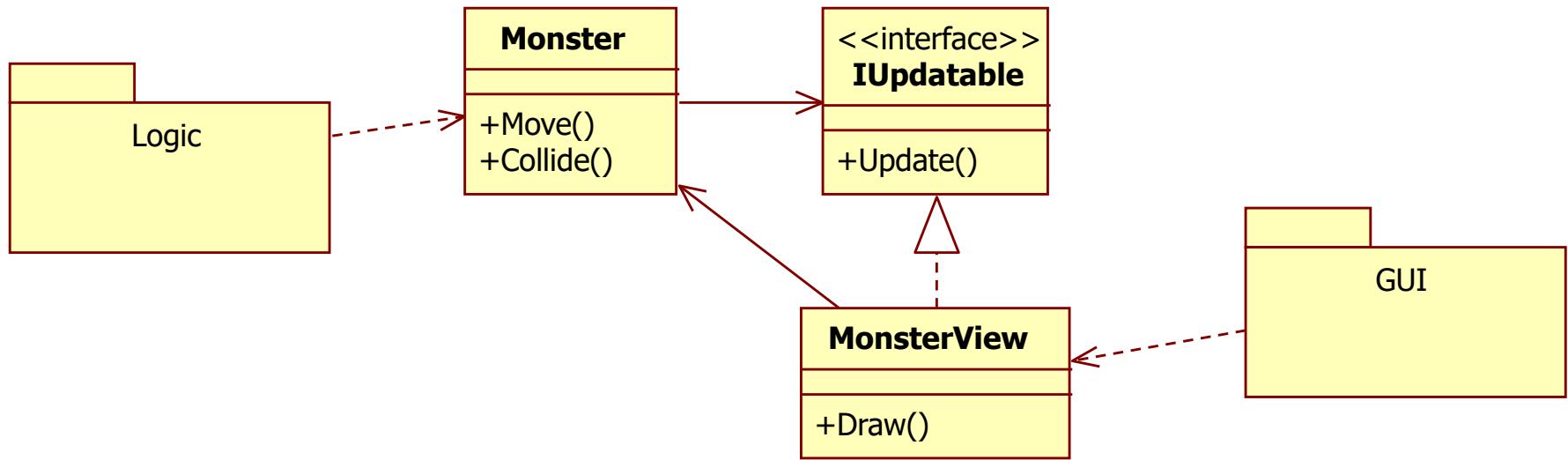
SRP violation example



SRP solution I.



SRP solution II.



Probability of change

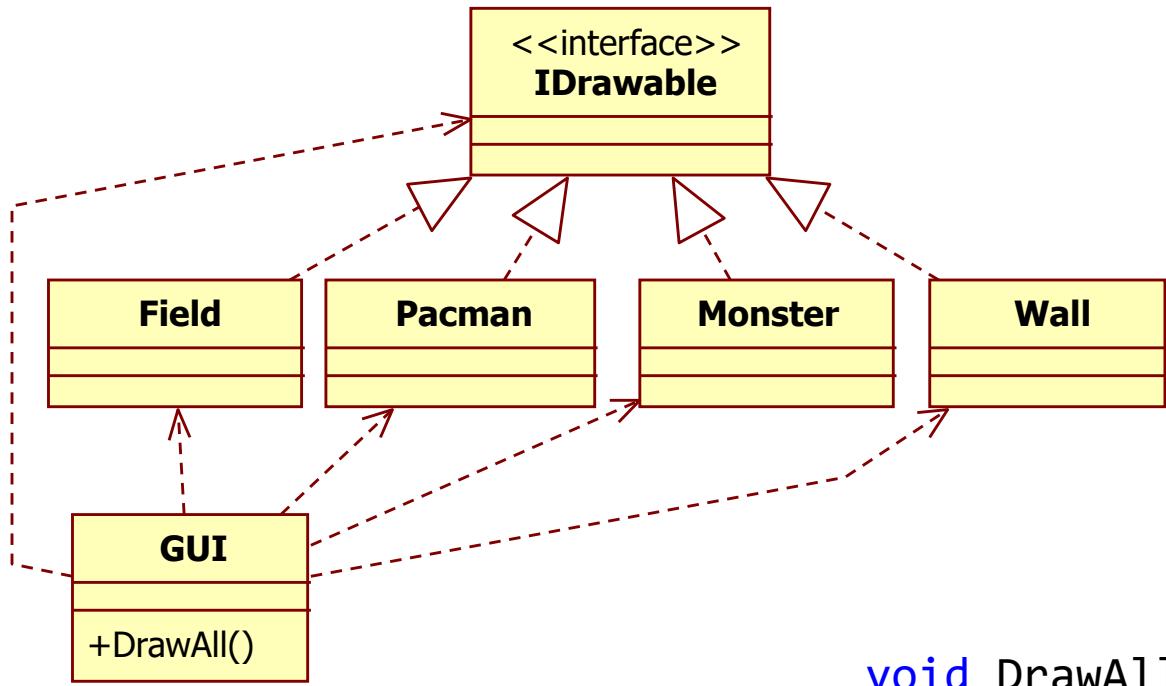
- It is not always obvious that there are more reasons to change
- It is also possible that the change will never occur
- We have to estimate the probability of the change based on our past experiences and on our domain knowledge
- Don't design for change if it has low probability
 - YAGNI = You Ain't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

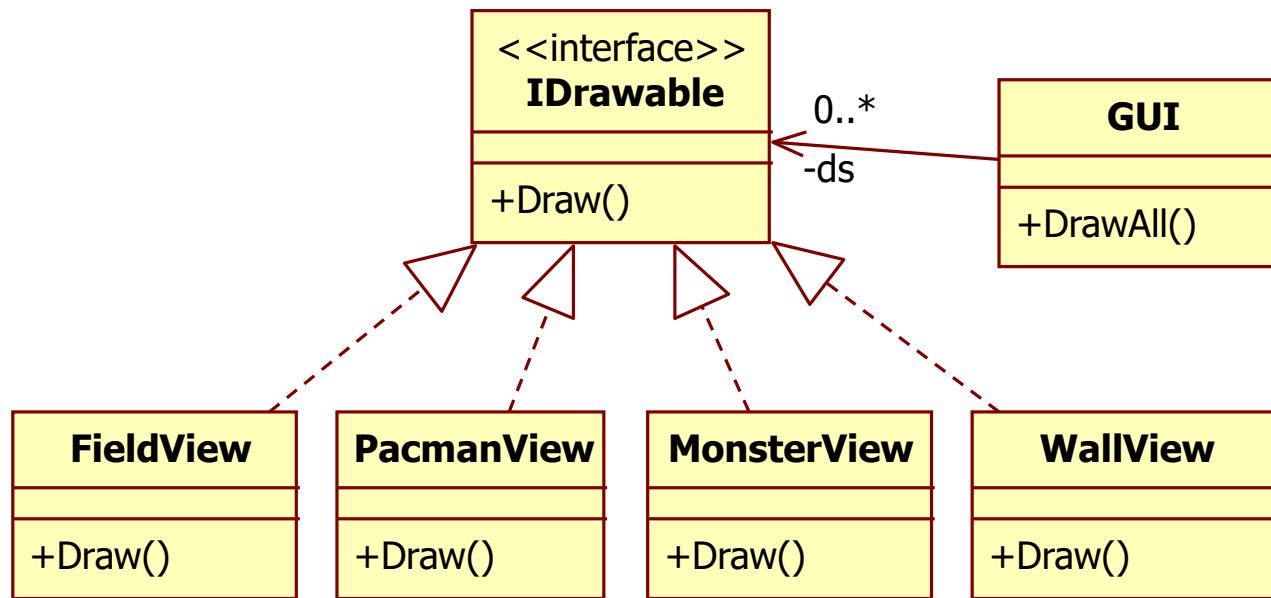
- Prepare for change
- Open for extension:
 - new subclasses
 - overriding methods
 - polymorphism
 - delegation
- Closed for modification:
 - original code does not change
 - only bugfixes
- Extend behavior by adding new code, not changing existing code

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 ...
    }
}
```

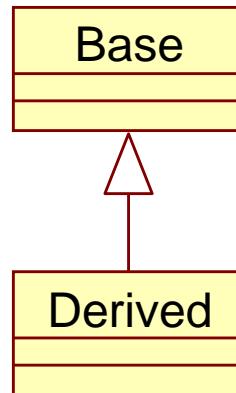
OCP solution



```
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 wherever 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



- 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

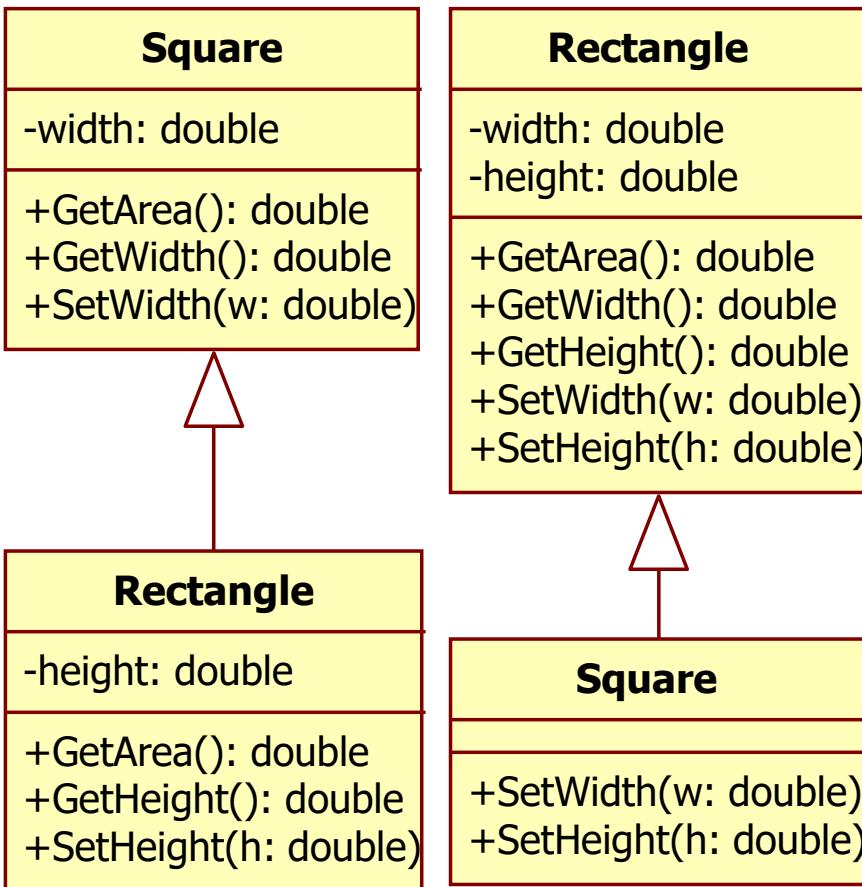
LSP violation: type check

- 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

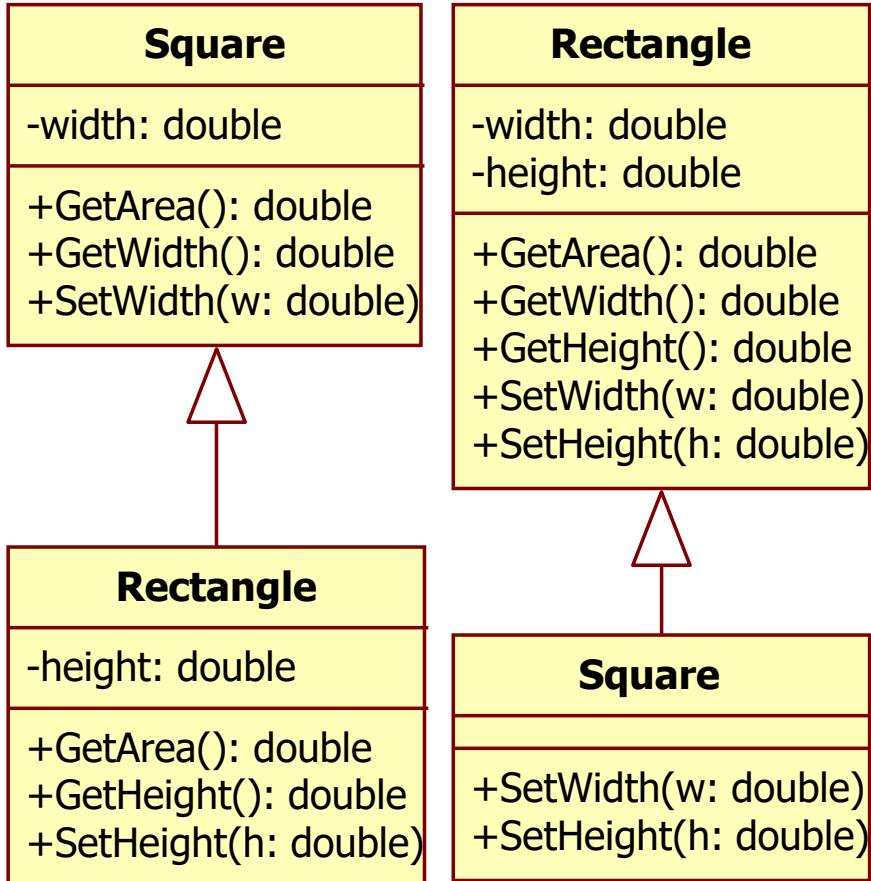
Which design is better?



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);
}
```

- 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

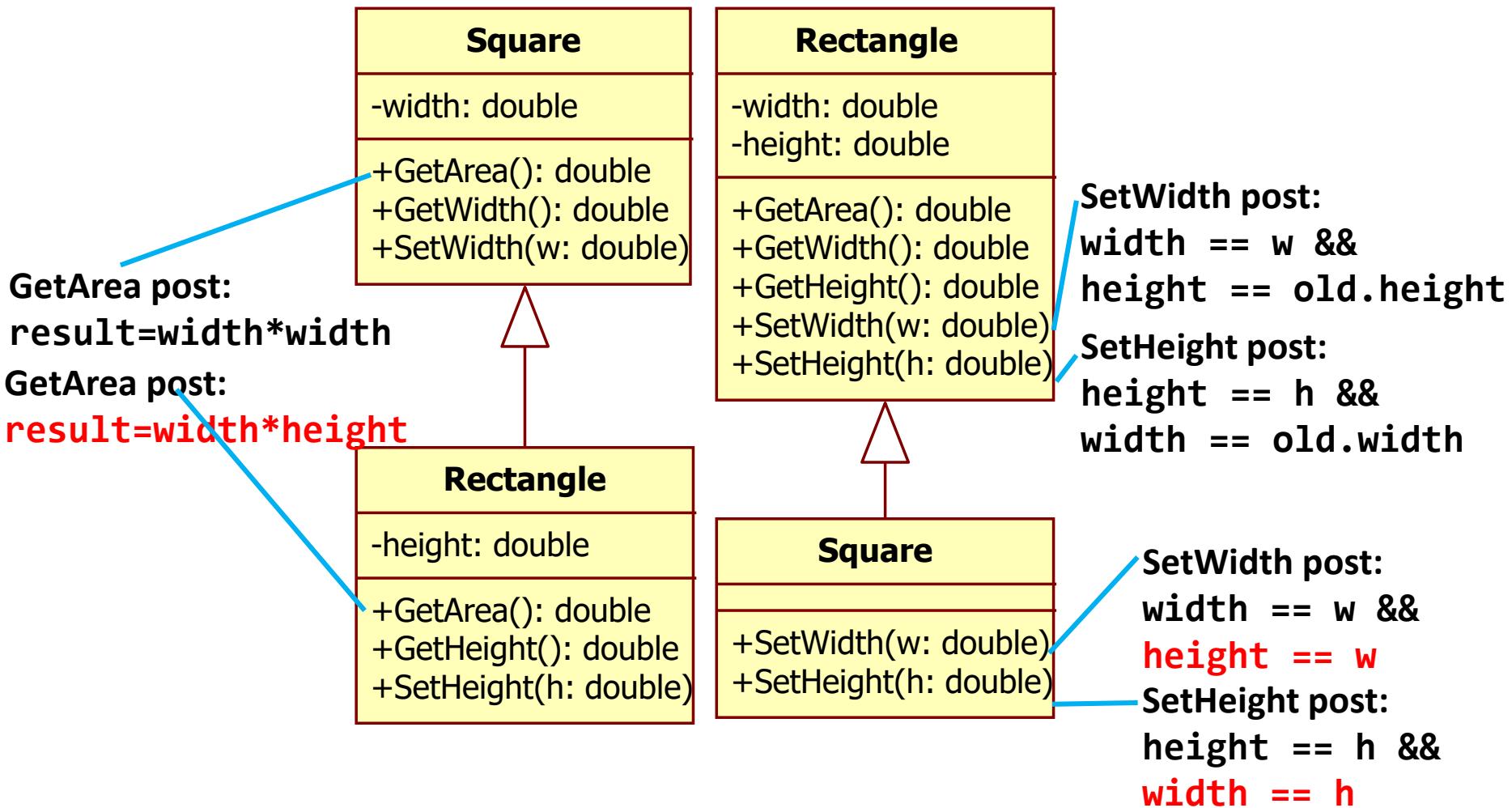
- Class invariants:

- 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 1 and 31
 - 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

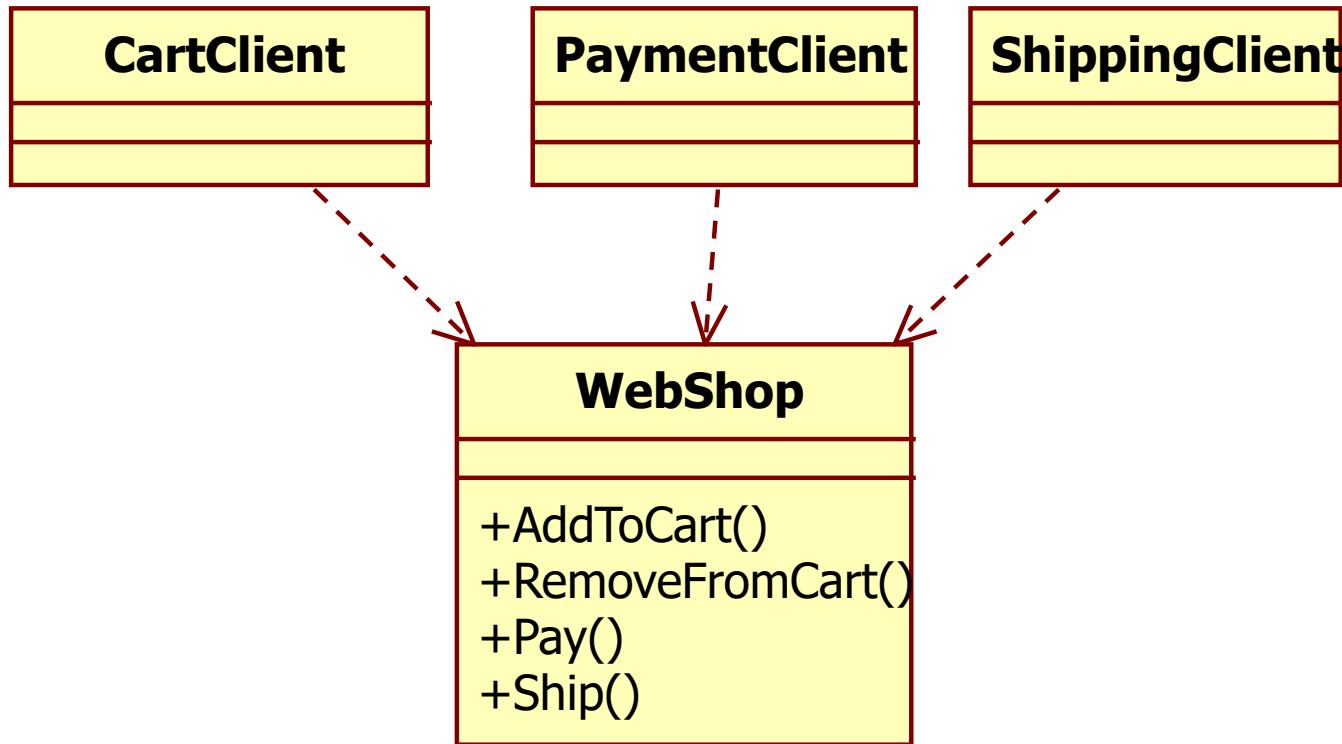
Violation of post-conditions



Interface Segregation Principle (ISP)

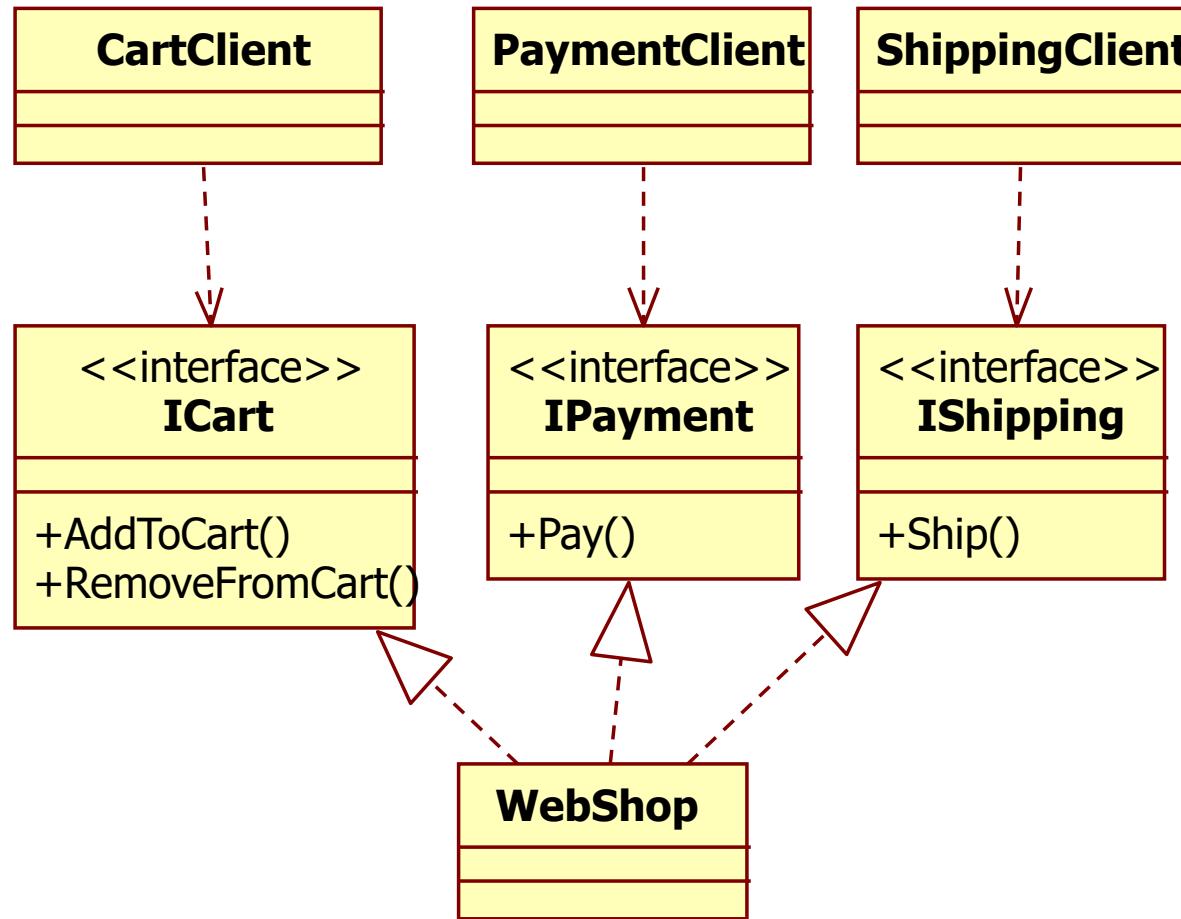
- Clients should not be forced to depend on methods they do not use
 - (Robert C. Martin)
- ISP acknowledges that objects require fat, non-cohesive interfaces
- However, clients should not know about them as a single class
- Instead, clients should know about abstractions with cohesive interfaces

ISP violation example



- 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

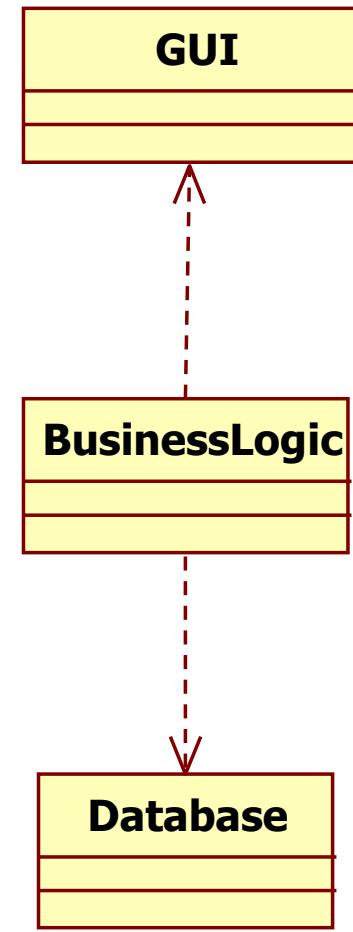


Dependency Inversion Principle (DIP)

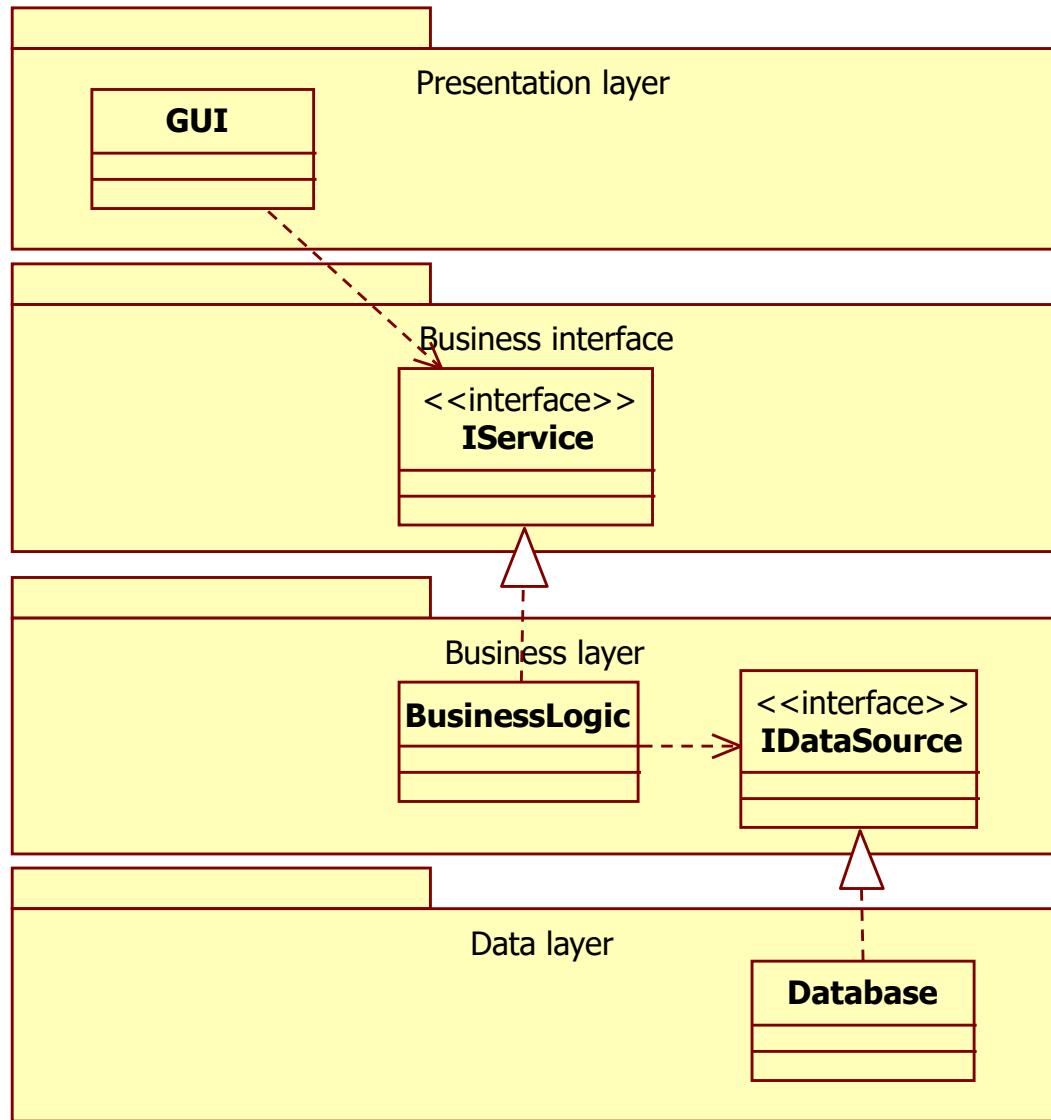
- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.
 - (Robert C. Martin)
- An application consists of modules
- A natural way to implement these modules is to write the low-level 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



- 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)

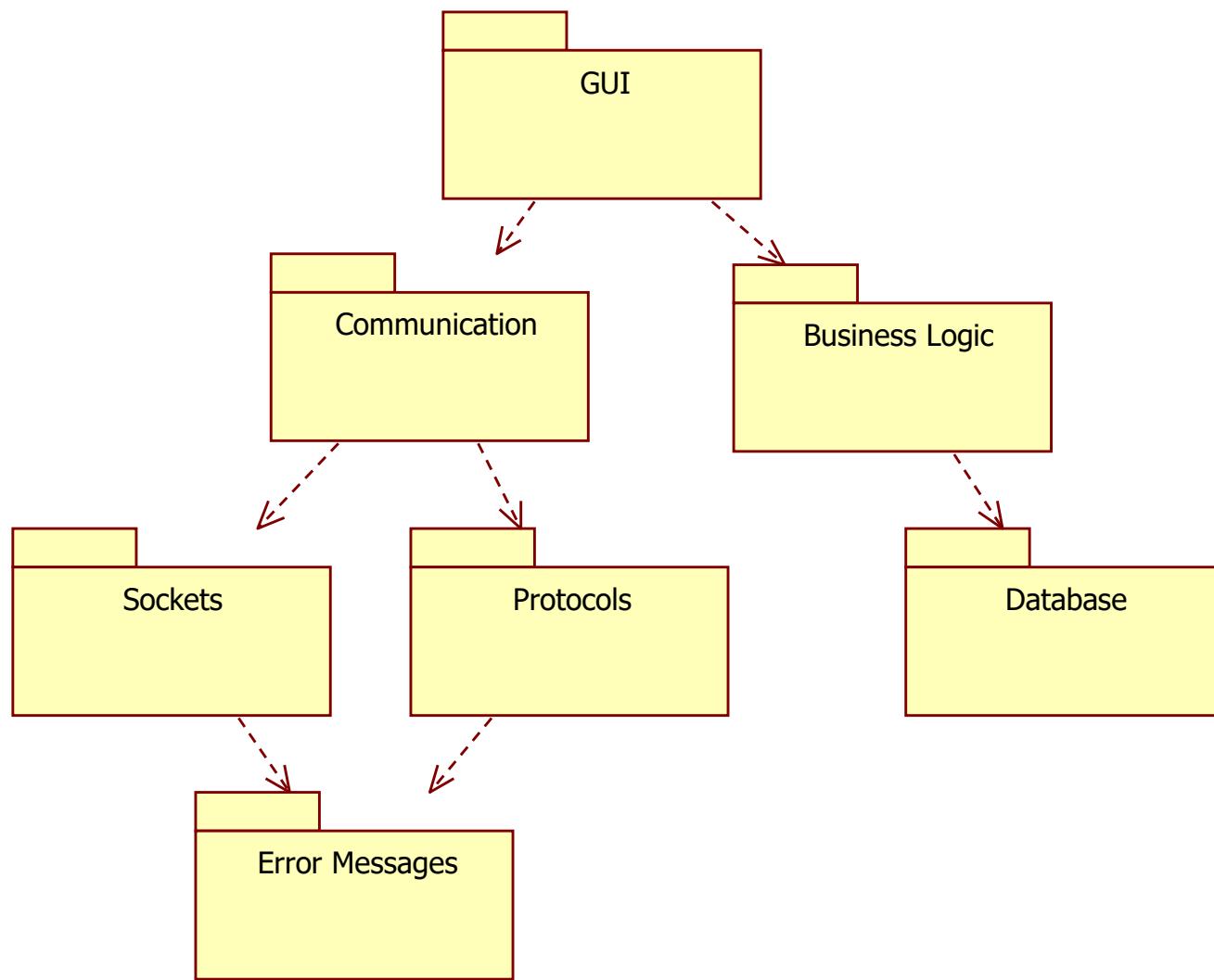
- Classes that change together, belong together
- The more packages that change in a release, the greater the work is to adjust, rebuild, test and deploy the new release
- Thus, classes that are likely to change together should be grouped into the same package
- We have to anticipate the possible changes,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

Common Reuse Principle (CRP)

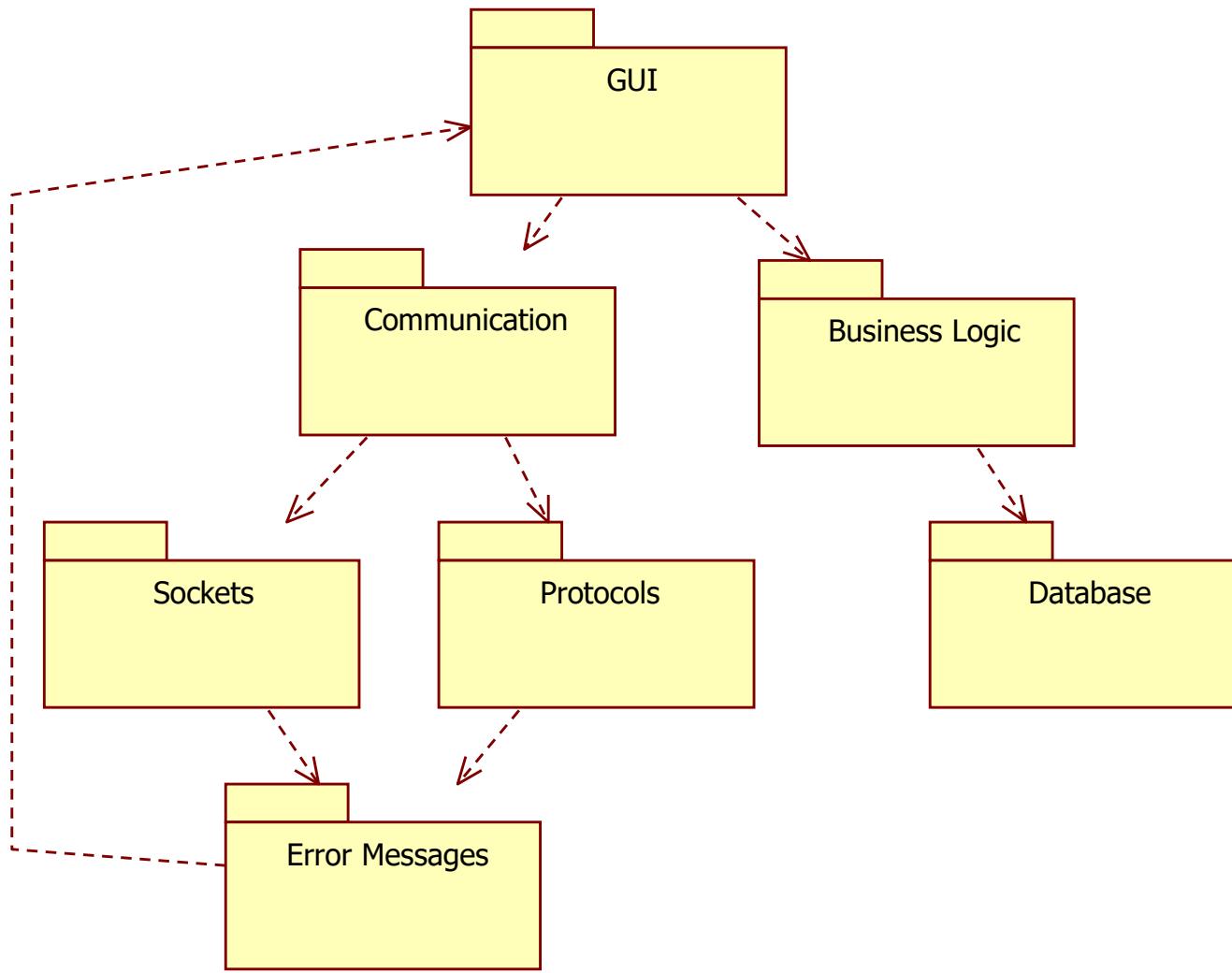
- Classes that aren't reused together should not be grouped together
- Dependency upon a package is a dependency upon everything in the package
- When a package changes, all its users have to be changed, even if they do not use the changed elements from the package
- This is similar to ISP at package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

Acyclic Dependencies Principle (ADP)

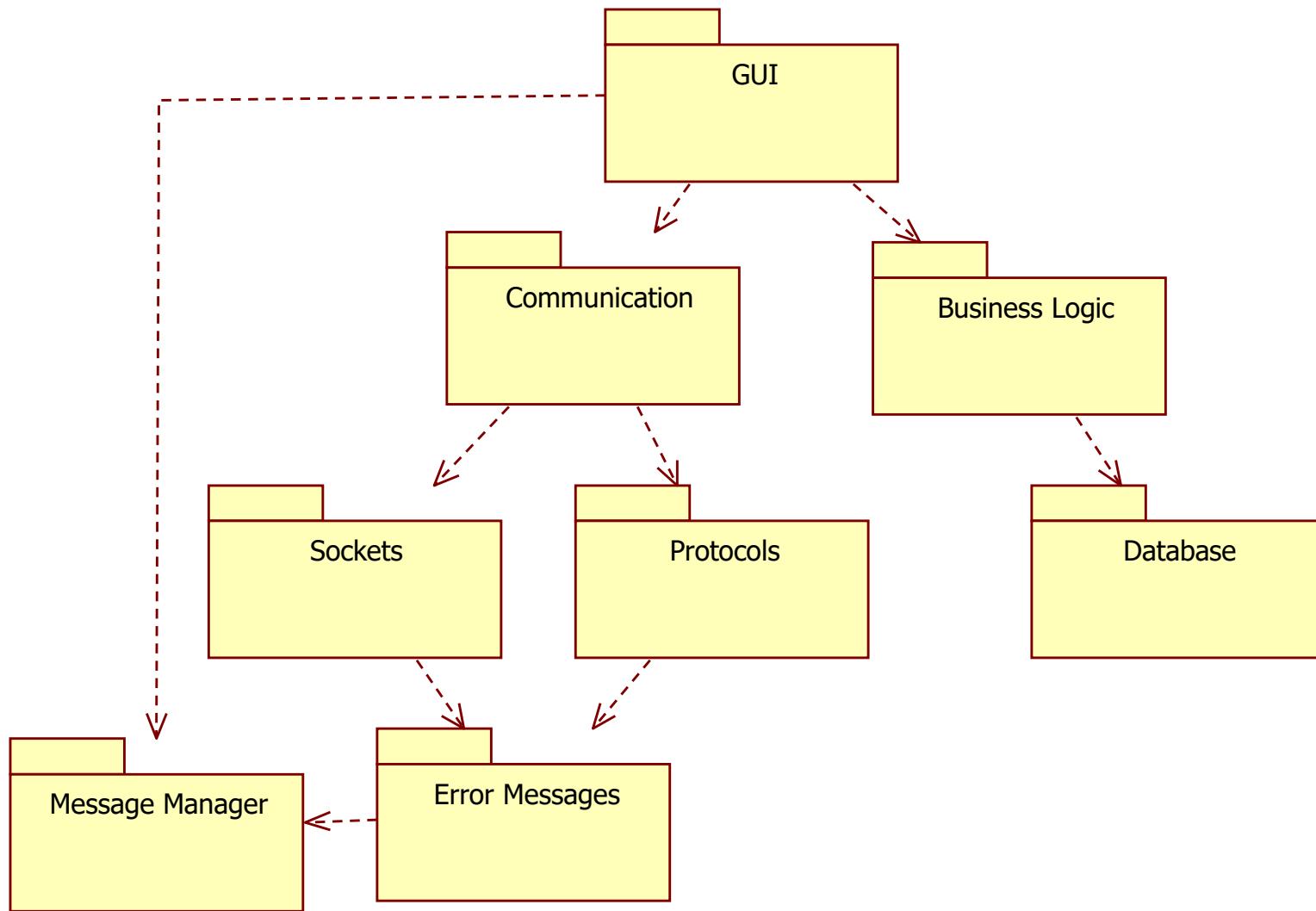
- Dependencies between packages must not form cycles
- A newly released package must be tested with its dependencies
- Hopefully, the number of dependencies is small
- But if there is a cycle in the dependency graph, all the packages in the cycle must be rebuilt and retested



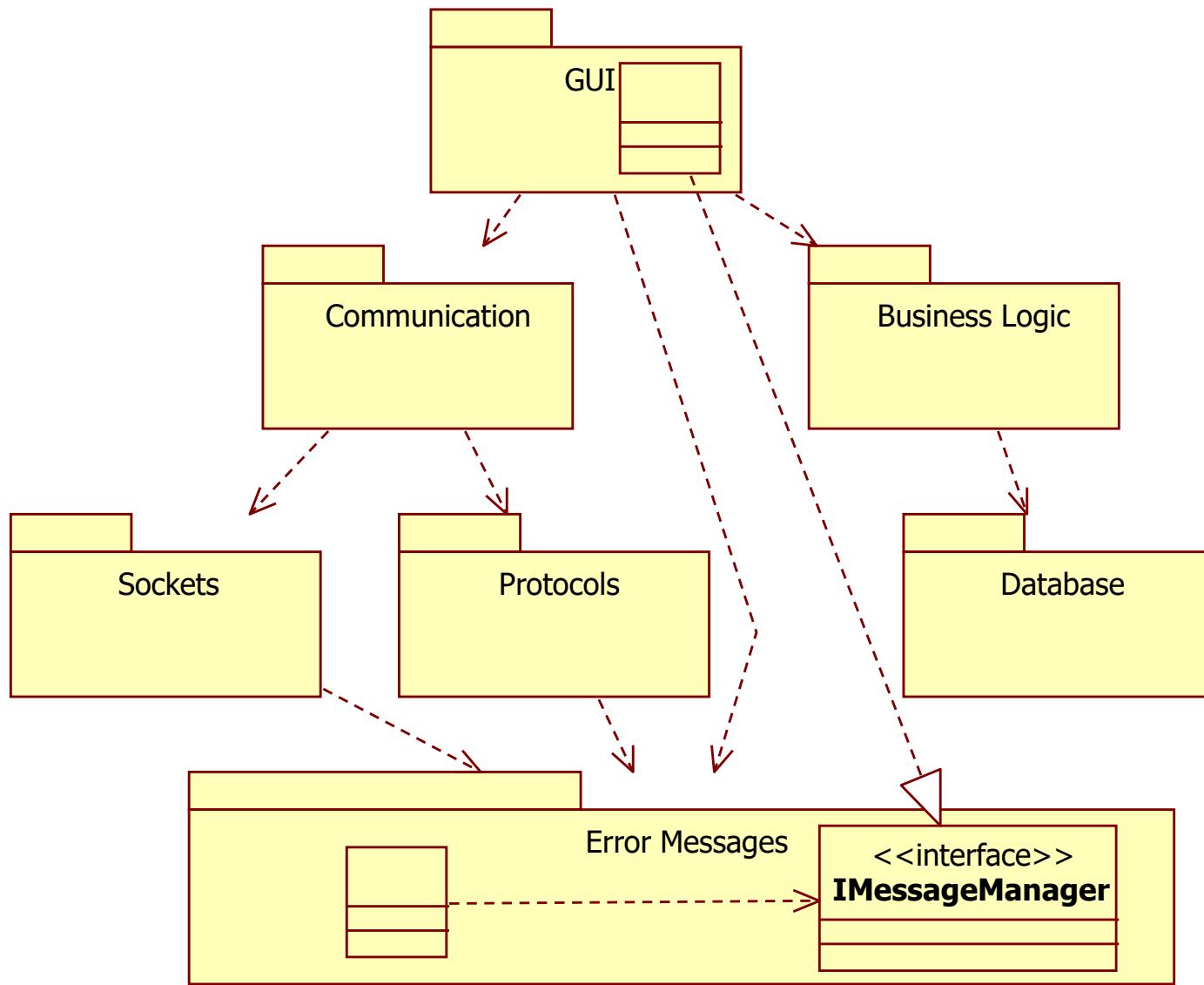
ADP violation example



ADP solution I: introducing a new package

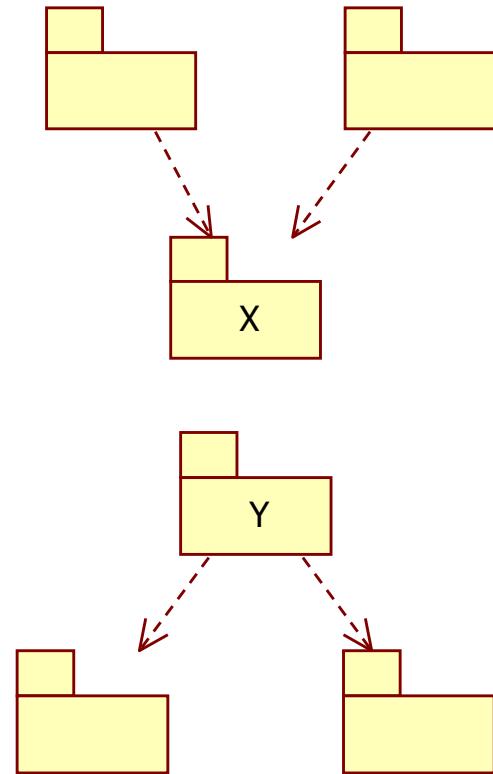


ADP solution II: DIP and ISP



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



- 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 to be difficult to extend!

- 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 thread-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 thread-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;  
        }  
    }  
}
```

- 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();
```

- 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)

Object-oriented design heuristics

Objektumorientált szoftvertervezés

Object-oriented software design

Dr. Balázs Simon

BME, IIT

Outline

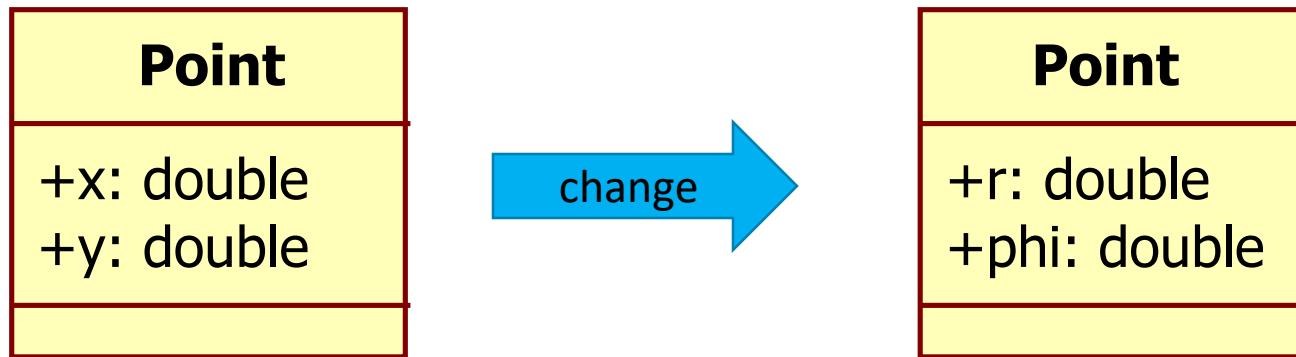
- Classes
- Responsibilities
- Associations
- Inheritance

Classes

Views of a class

- Creator:
 - creates the class
 - provides the implementation
 - hides the implementation details in case it will change
 - exports only the strictly useful details to the users
- Users:
 - client programmers
 - use the public interface
 - see only what is important to them
 - don't see what they shouldn't see
- Developers are usually in both hats at the same time

C1. Problem



Problem: public attributes

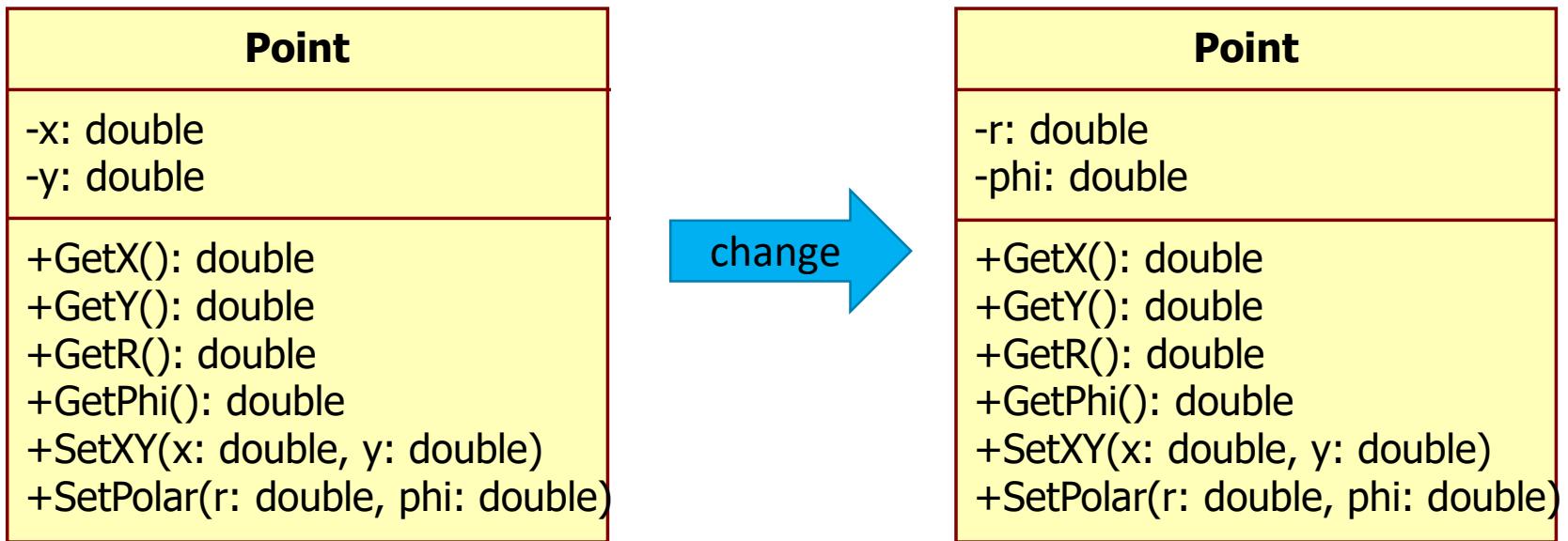
C1. Attributes should be private

- Problems if violated:
 - protected and public attributes violate information hiding
 - maintenance issues if data representation changes
 - constraints have to be checked everywhere
 - violates DRY
 - inner state can be broken

C1. Attributes should be private

- Rule:
 - attributes should always be private
 - if a public/protected attribute would be necessary for some external operation, the class should provide that operation
 - the state of an object should always be changed through its public methods
- Exceptions:
 - static constant attributes can be public

C1. Solution



C2. Problem

```
public class Point {  
    internal double x;  
    internal double y;  
    ...  
    public Point(double x, double y) { ... }  
    ...  
};  
  
public class Serializer {  
    private StreamWriter w;  
    ...  
    public void Write(Point p) {  
        w.WriteLine(string.Format("{0}, {1}", p.x, p.y));  
    }  
}
```

Problem: accessing non-public members of another class

C2. Do not use non-public members of another class

- Problems if violated:

- if package/internal/friend is used to access non-public members because the public interface of the target class is insufficient or incomplete
- using non-public members of another class increases coupling because the user depends on the implementation details of the target class

C2. Do not use non-public members of another class

- Rule:
 - do not use non-public members of another class
 - check if the public interface of the target class could be modified to be complete for the task
 - check whether the two classes should be one, maybe the responsibilities should belong to a single class
- Exceptions:
 - if you are writing a library/framework and you intentionally want to hide certain parts from the users
 - e.g. allow instantiation of objects only through a factory class
 - testing

C2. Solution

```
public class Point {  
    private double x;  
    private double y;  
    ...  
    public Point(double x, double y) { ... }  
    ...  
    public double X { get { return x; } }  
    public double Y { get { return y; } }  
};  
  
public class Serializer {  
    private StreamWriter w;  
    ...  
    public void Write(Point p) {  
        w.WriteLine(string.Format("{0}, {1}", p.X, p.Y));  
    }  
}
```

C3. Problem: java.awt.Label

Methods inherited from class java.awt.Component

```
action, add, addComponentListener, addFocusListener, addHierarchyBoundsListener, addHierarchyListener,
addInputMethodListener, addKeyListener, addMouseListener, addMouseMotionListener, addMouseWheelListener,
addPropertyChangeListener, addPropertyChangeListener, applyComponentOrientation, areFocusTraversalKeysSet, bounds,
checkImage, checkImage, coalesceEvents, contains, contains, createImage, createImage, createVolatileImage,
createVolatileImage, deliverEvent, disable, disableEvents, dispatchEvent, doLayout, enable, enable, enableEvents,
enableInputMethods, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange,
firePropertyChange, firePropertyChange, firePropertyChange, firePropertyChange, getAlignmentX, getAlignmentY,
getBackground, getBaseline, getBaselineResizeBehavior, getBounds, getBounds, getColorModel, getComponentAt, getComponentAt,
getComponentListeners, getComponentOrientation, getCursor, getDropTarget, getFocusCycleRootAncestor, getFocusListeners,
getFocusTraversalKeys, getFocusTraversalKeysEnabled, getFont, getFontMetrics, getForeground, getGraphics,
getGraphicsConfiguration, getHeight, getHierarchyBoundsListeners, getHierarchyListeners, getIgnoreRepaint, getInputContext,
getInputMethodListeners, getInputMethodRequests, getKeyListeners, getListeners, getLocale, getLocation, getLocation,
getLocationOnScreen, getMaximumSize, getMinimumSize, getMouseListeners, getMouseMotionListeners, getMousePosition,
getMouseWheelListeners, getName, getParent, getPeer, getPreferredSize, getPropertyChangeListeners,
getPropertyChangeListeners, getSize, getSize, getToolkit, getTreeLock, getWidth, getX, getY, gotFocus, handleEvent,
hasFocus, hide, imageUpdate, inside, invalidate, isBackgroundSet, isCursorSet, isDisplayable, isDoubleBuffered, isEnabled,
isFocusable, isFocusCycleRoot, isFocusOwner, isFocusTraversable, isFontSet, isForegroundSet, isLightweight,
isMaximumSizeSet, isMinimumSizeSet, isOpaque, isPreferredSizeSet, isShowing, isValid, isVisible, keyDown, keyUp, layout,
list, list, list, locate, location, lostFocus, minimumSize, mouseDown, mouseDrag, mouseEnter, mouseExit,
mouseMove, mouseUp, move, nextFocus, paint, paintAll, postEvent, preferredSize, prepareImage, prepareImage, print,
printAll, processComponentEvent, processEvent, processFocusEvent, processHierarchyBoundsEvent, processHierarchyEvent,
processInputMethodEvent, processKeyEvent, processMouseEvent, processMouseMotionEvent, processMouseWheelEvent, remove,
removeComponentListener, removeFocusListener, removeHierarchyBoundsListener, removeHierarchyListener,
removeInputMethodListener, removeKeyListener, removeMouseListener, removeMouseMotionListener, removeMouseWheelListener,
removeNotify, removePropertyChangeListener, removePropertyChangeListener, repaint, repaint, repaint, repaint, requestFocus,
requestFocus, requestFocusInWindow, requestFocusInWindow, reshape, resize, resize, setBackground, setBounds, setBounds,
setComponentOrientation, setCursor, setDropTarget, setEnabled, setFocusable, setFocusTraversalKeys,
setFocusTraversalKeysEnabled, setFont, setForeground, setIgnoreRepaint, setLocale, setLocation, setLocation,
setMaximumSize, setMinimumSize, setName, setPreferredSize, setSize, setSize, setVisible, show, show, size, toString,
transferFocus, transferFocusBackward, transferFocusUpCycle, update, validate
```

Problem: too many public methods

C3. Keep the number of public methods in a class minimal

- Problems if violated:

- the user does not need too much methods
- hard to find the method we are looking for
 - e.g. operator+ instead of union
- there may be multiple ways to do the same thing
- making private or protected methods public reveals too much about the implementation

C3. Keep the number of public methods in a class minimal

- Rule:

- minimize the number of public methods
- do not publish private or protected methods
- do not clutter the public interface with items that are not intended to be used (ISP)
- provide only one way to do things

C4. Problem

```
public class SyntaxNode
{
    public SyntaxNode(string name) { }
    public SyntaxNode(string name, params SyntaxNode[] children) { }
    public string Name { get; }
    public ImmutableArray<SyntaxNode> Children { get; }
};

...
Problem: missing standard operations

public static void Main(string[] args)
{
    SyntaxNode n1 = new SyntaxNode("n1", 0);
    SyntaxNode n2 = new SyntaxNode("n2", n1);

    Console.WriteLine(n1);

    if (n1 == n2) { ... }

    var d = new Dictionary<SyntaxNode, Diagnostic>();
    d.Add(n1, new Diagnostic("Error message for node 1."));
}
```

C4. Implement a minimal set of methods in all classes

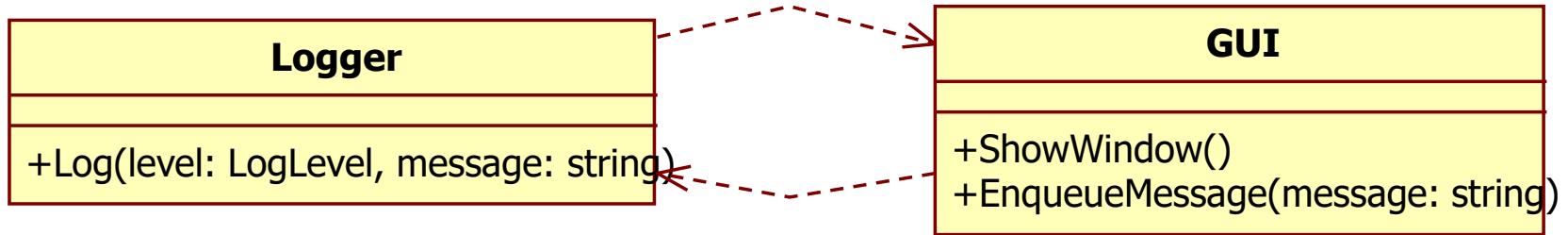
- Useful methods:
 - equality testing, printing, parsing, deep copy
 - C#: ToString(), Equals(), GetHashCode()
 - Java: toString(), equals(), hashCode()
 - C++: copy constructor, operator==, operator=, operator<< to an ostream
- These are useful since the developer can expect these to work
- They can also be useful for testing

C4. Solution

```
public class SyntaxNode
{
    public SyntaxNode(string name) { }
    public SyntaxNode(string name, params SyntaxNode[] children) { }
    public string Name { get; }
    public ImmutableList<SyntaxNode> Children { get; }

    public override string ToString()
    {
        return this.Name;
    }
    public override int GetHashCode()
    {
        return this.Name.GetHashCode();
    }
    public override bool Equals(object obj)
    {
        SyntaxNode node = obj as SyntaxNode;
        if (node == null) return false;
        return node.Name == this.Name;
    }
}
```

C5. Problem



Problem: circular dependency

C5. A class should not depend on its users

- Problems if violated:

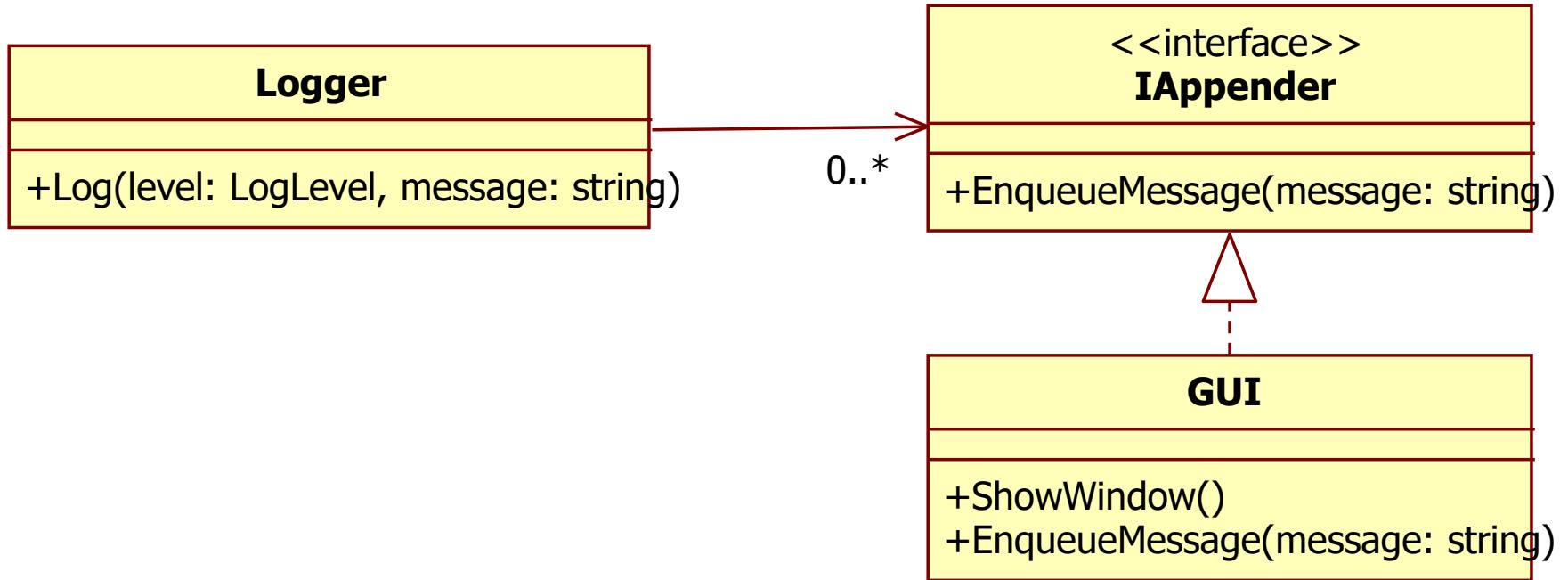
- if a class depends on its users it cannot be reused without them
 - descendants already depend on the base class
 - if a class depends on its descendants it has to depend on later added descendants, too
 - this violates OCP and LSP
 - also this creates circular dependencies
 - this violates ADP

C5. A class should not depend on its users

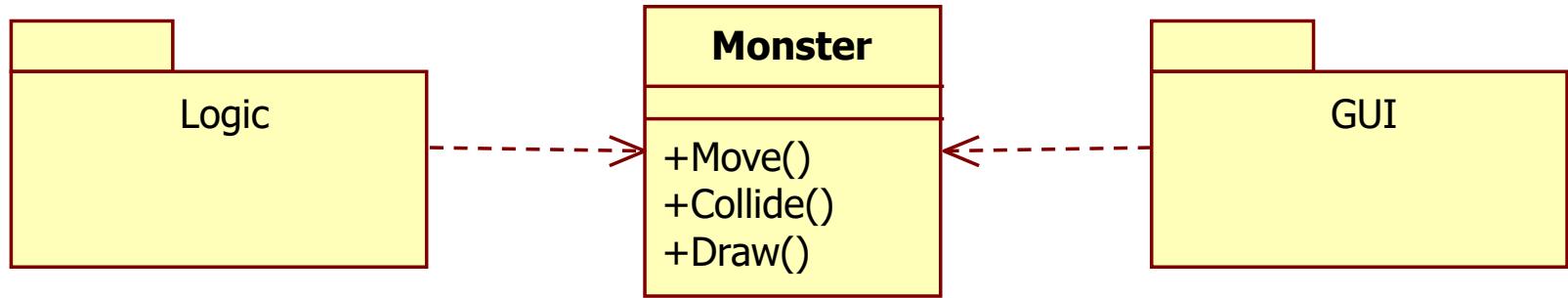
- Rule:

- a class must not know about its descendants
- if a class depends on its users try to minimize this dependency (ISP) or resolve (DIP) it completely
- also check if the responsibility divided between the class and its users should really be divided or it should be transferred to a single class (see next rules: C6, C7)

C5. Solution



C6. Problem



Problem: too many responsibilities in a single class

C6. A class should capture exactly one abstraction

- Here abstraction means responsibility
- This is a corollary to SRP
- Problems if violated:
 - if a class captures more than one abstraction, it violates SRP
 - it has more reasons to change
 - if an abstraction is distributed between more than one class it may violate DRY

C6. A class should capture exactly one abstraction

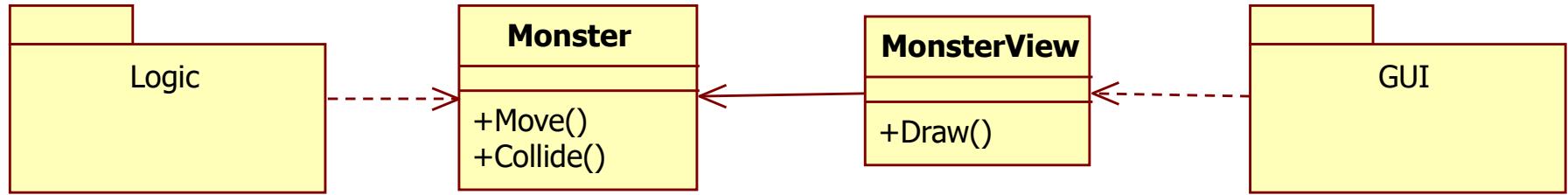
- Rule:

- if a class captures more than one responsibilities, consider to split it up into multiple classes, one for each responsibility
 - if it cannot be split up, use ISP
 - if a responsibility is divided by violating DRY, move that violating behavior to a single class

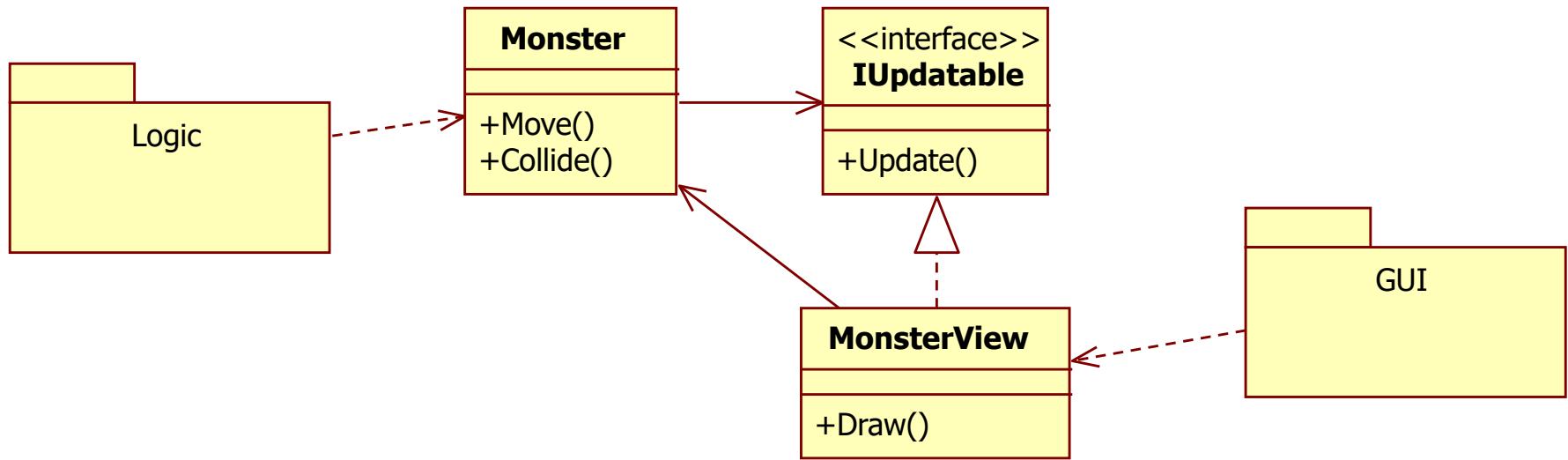
- Exception:

- if a responsibility is divided but DRY is not violated, it is perfectly OK
 - e.g. Design Patterns: visitor, strategy

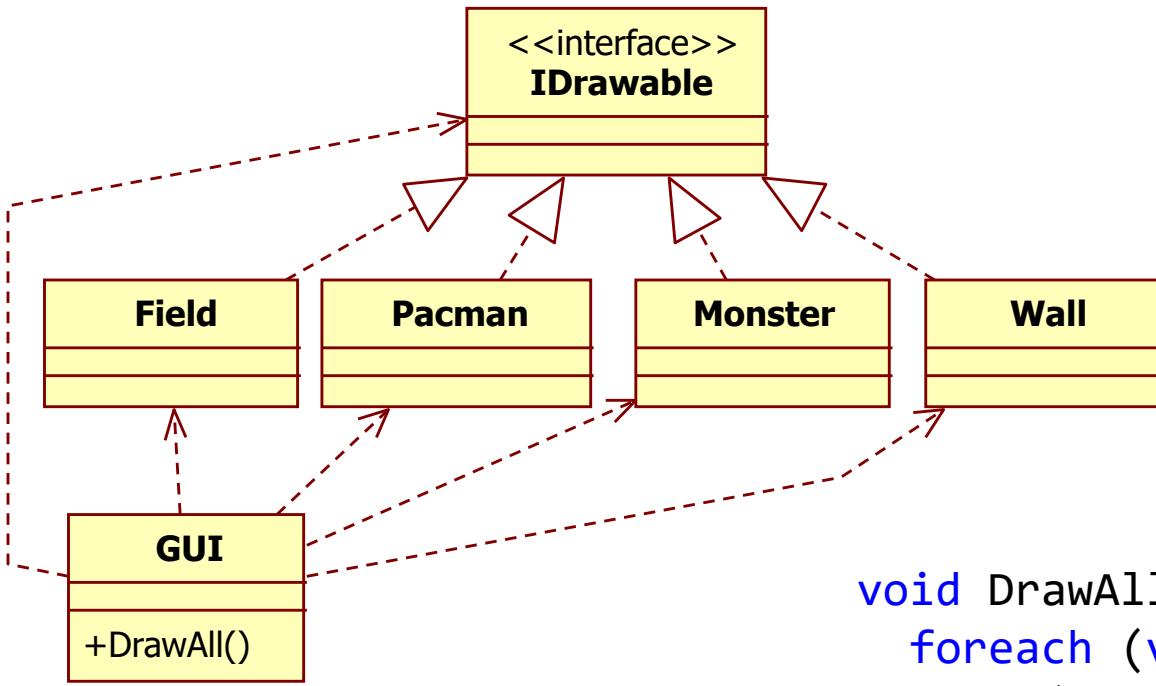
C6. Solution I.



C6. Solution II.



C7. Problem



```
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 ...
    }
}
```

Problem: behavior is separated from the data

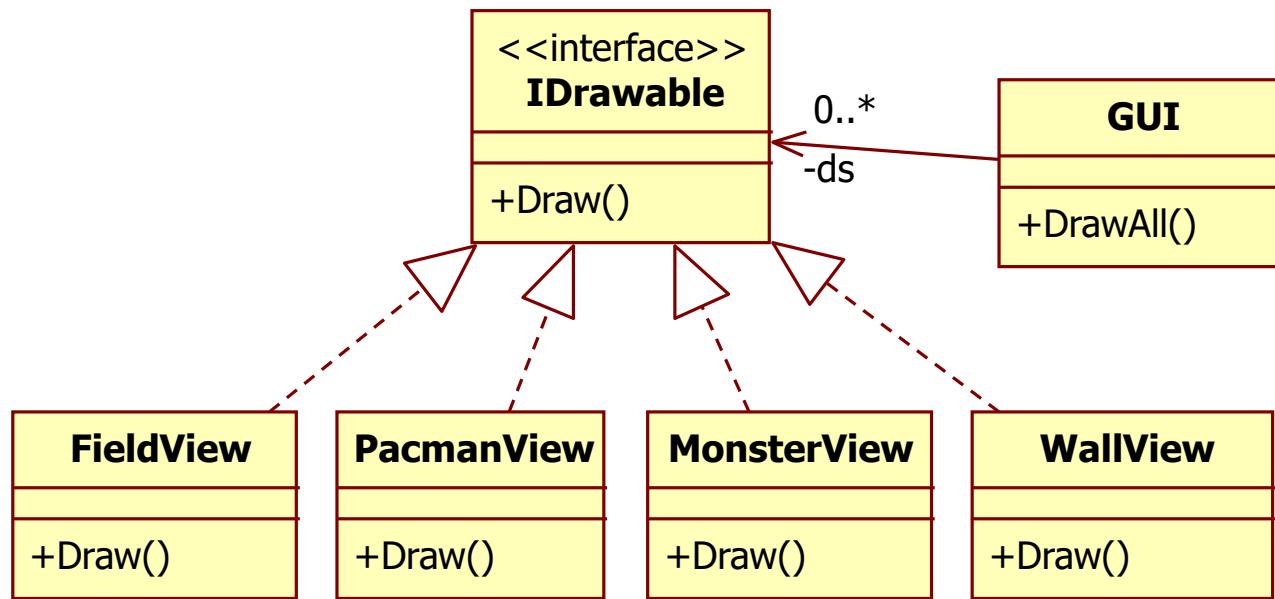
C7. Keep related data and behavior together

- Problems if violated:
 - behavior is separated from the data
 - to perform some operation two or more areas of the system are required
 - this usually leads to the violation of TDA and DRY
 - circular dependency may occur between the separated parts
 - procedural style:
 - a class for holding data with getters-setters
 - another class containing operations on the data

C7. Keep related data and behavior together

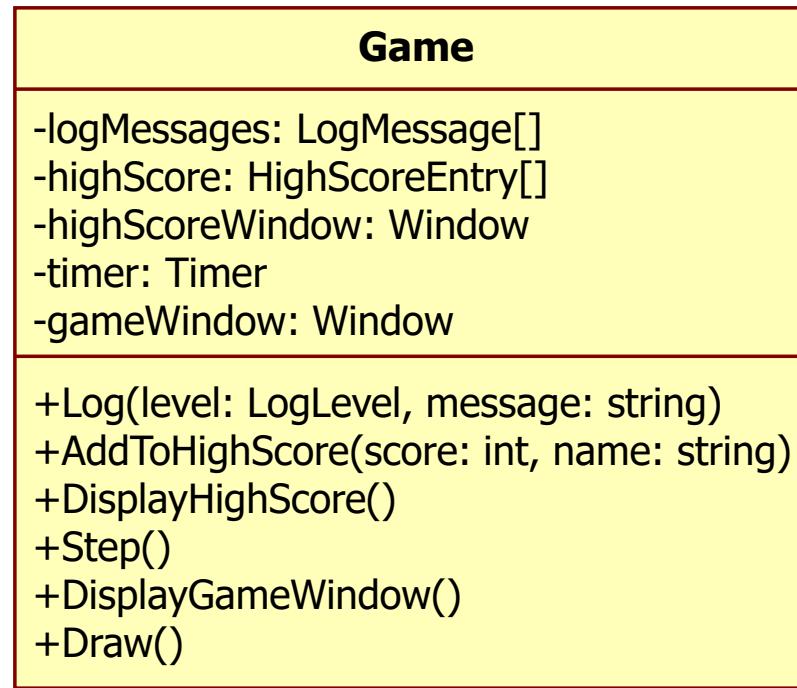
- Rule:
 - check for circular dependency between classes and combine them into a single class if they share a common responsibility
 - check for the violation of TDA and DRY, and combine the external behavior into a single class
 - check for data holder classes with only getters-setters and a separate class with operations on the data, and combine them into a single class
- Exception:
 - data holder classes for ORM and DTO are perfectly OK

C7. Solution



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

C8. Problem



Problem: too many and unrelated responsibilities

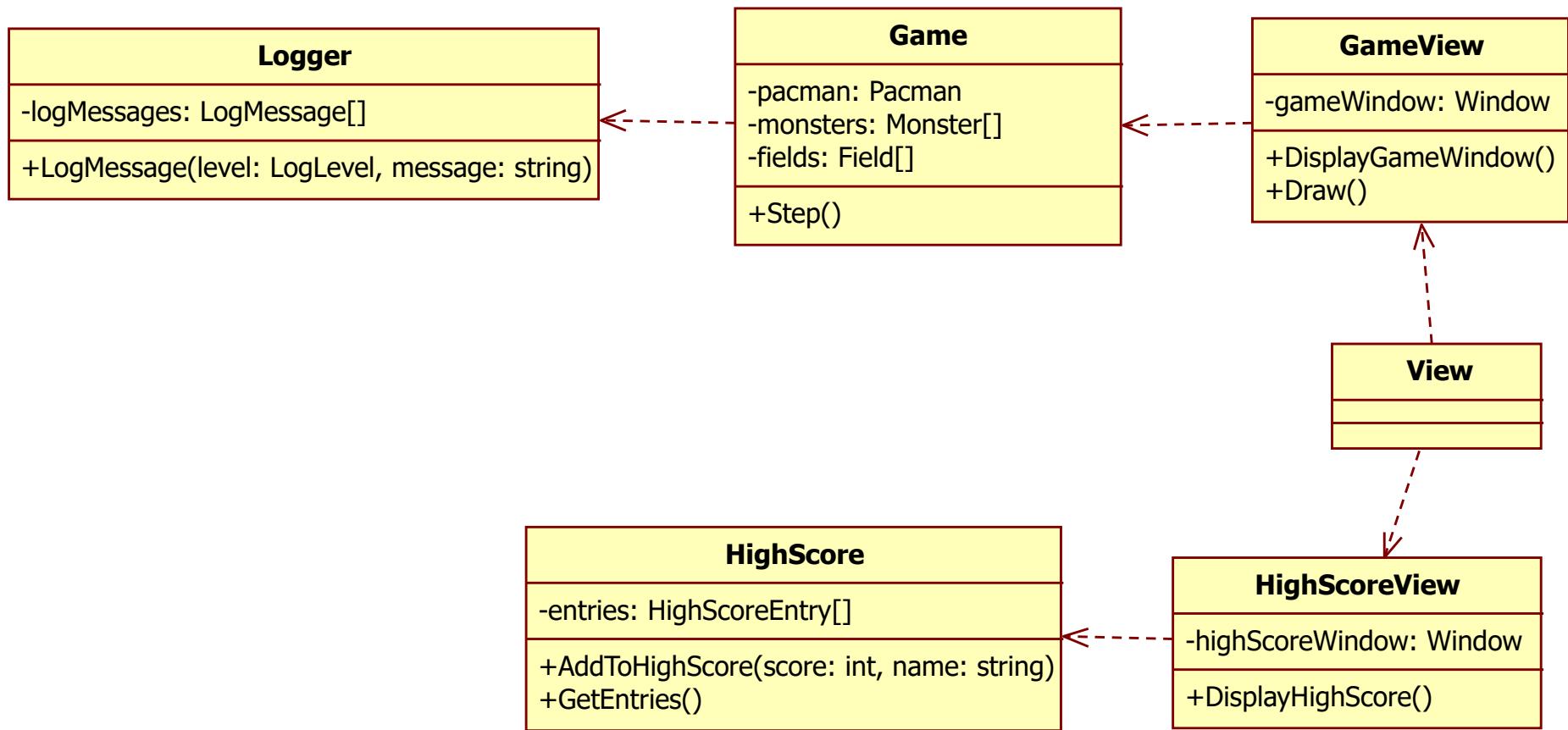
C8. Methods should use most of the members in its class

- Problems if violated:
 - the class is not cohesive enough
 - god class suspicion
 - it usually means that the class has more than one responsibilities: a violation of SRP

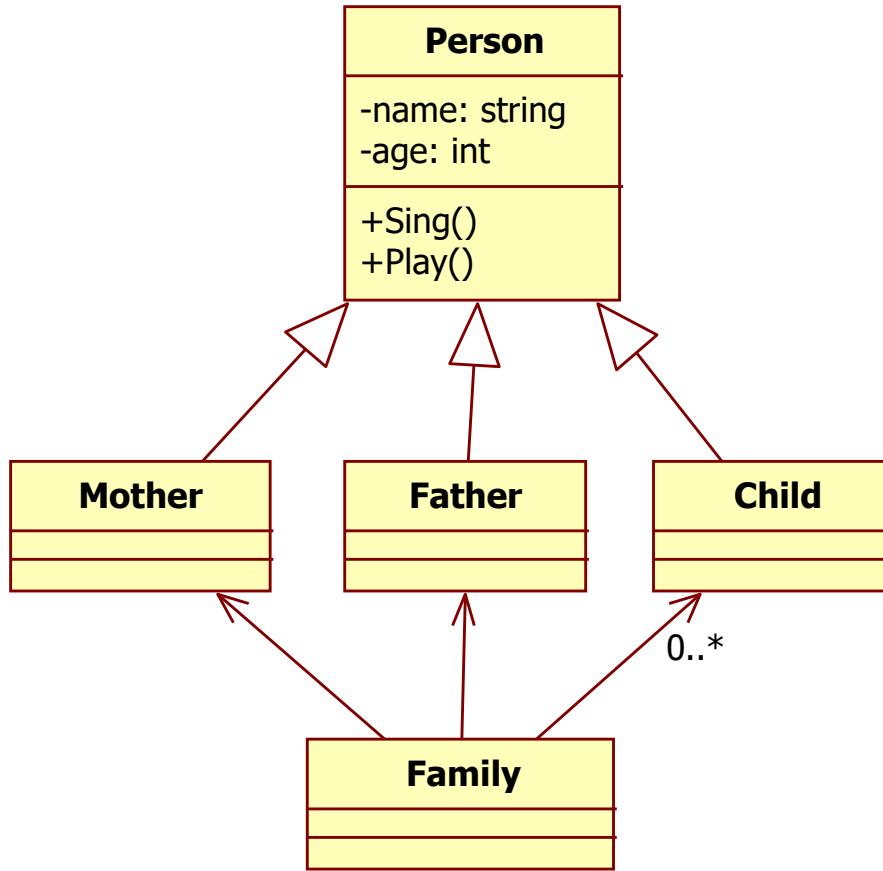
C8. Methods should use most of the members in its class

- Rule:
 - do not mix unrelated data and behavior in a single class
 - avoid god classes with methods of unrelated behavior
 - if this problem is found, split the class along the responsibilities into more cohesive classes
- Exceptions:
 - ORM and DTO classes, utility classes are OK

C8. Solution



C9. Problem



Problem: subclasses do not add anything to the behavior of the base class

C9. Model for behavior, not for roles

- Problems if violated:
 - the model contains classes based on their roles and not on their behavior
 - e.g. Mother and Father, which could be two instances of Person (of course, it depends on the domain)
 - there are descendant classes with no modified behavior
 - typically: leaves with no overriding methods in the inheritance hierarchy

C9. Model for behavior, not for roles

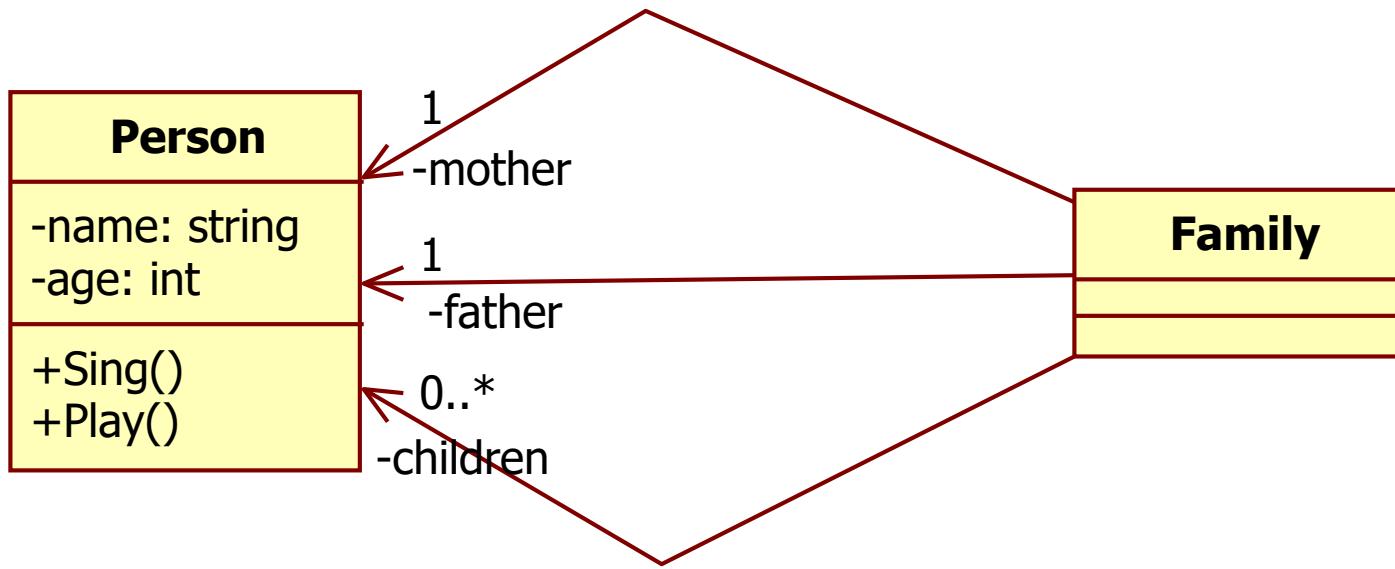
- Rules:

- model for behavior, not for roles
- if there are classes (usually leaves) with no overriding behavior in the inheritance hierarchy, they are probable signs for the violation of this rule
- if a member of the public interface cannot be used in a role, this implies the need for a different class
- if a member of the public interface is simply not being used (although it could be), then it is the same class with multiple roles

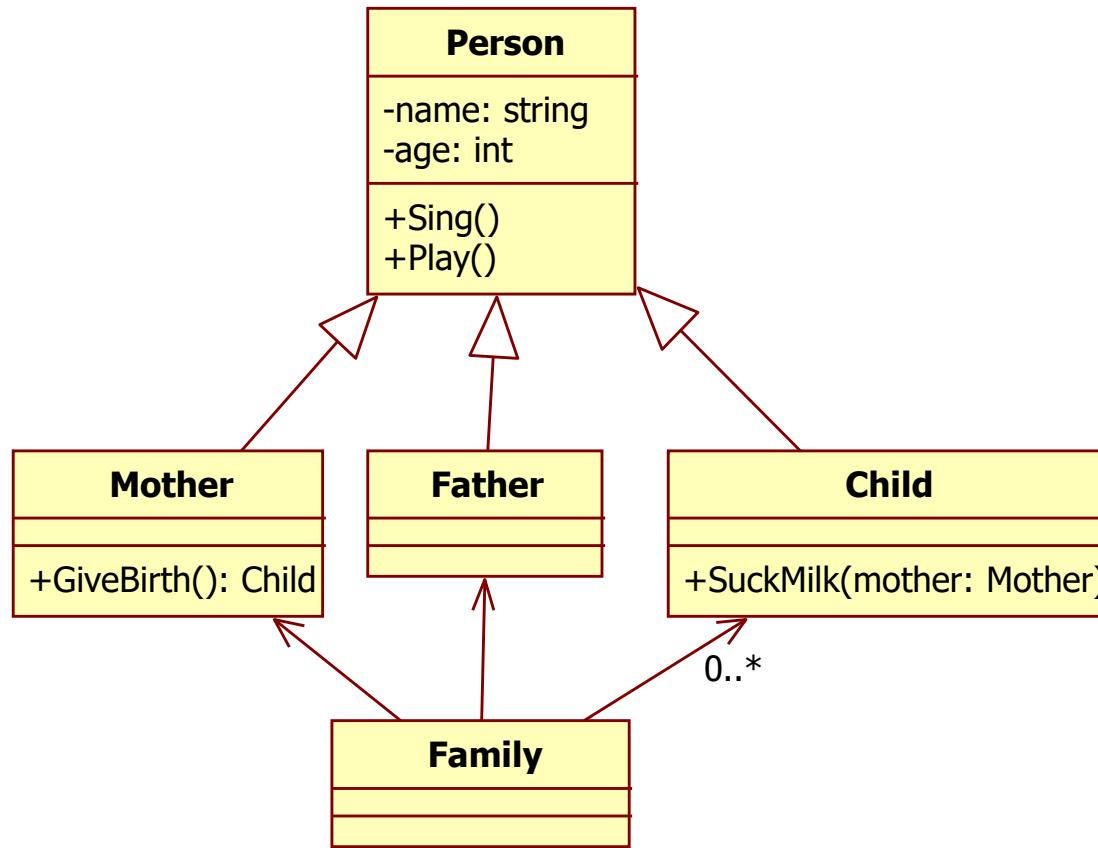
- Exceptions:

- empty leaves are allowed if behavior is separated from the class, e.g. visitor design pattern

C9. Solution I.



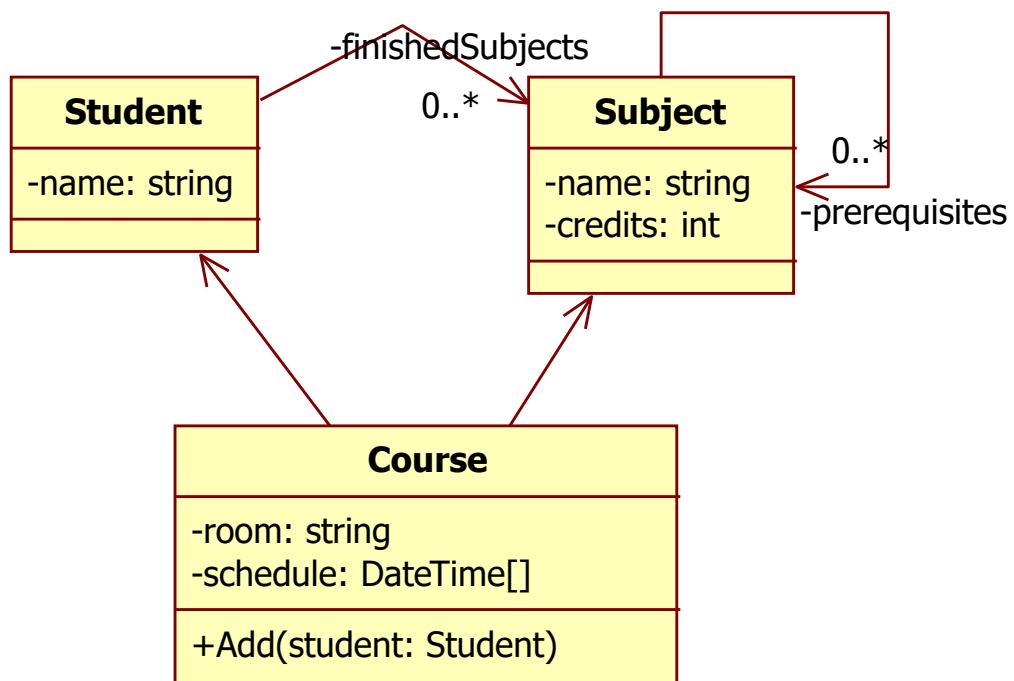
C9. Solution II.



Responsibilities

R1. Design problem

- Where to put the responsibility of checking whether the student fulfills the prerequisites of a subject?
 - in the Student?
 - in the Subject?
 - in the Course?
 - somewhere else?



R1. Distribute responsibility horizontally evenly

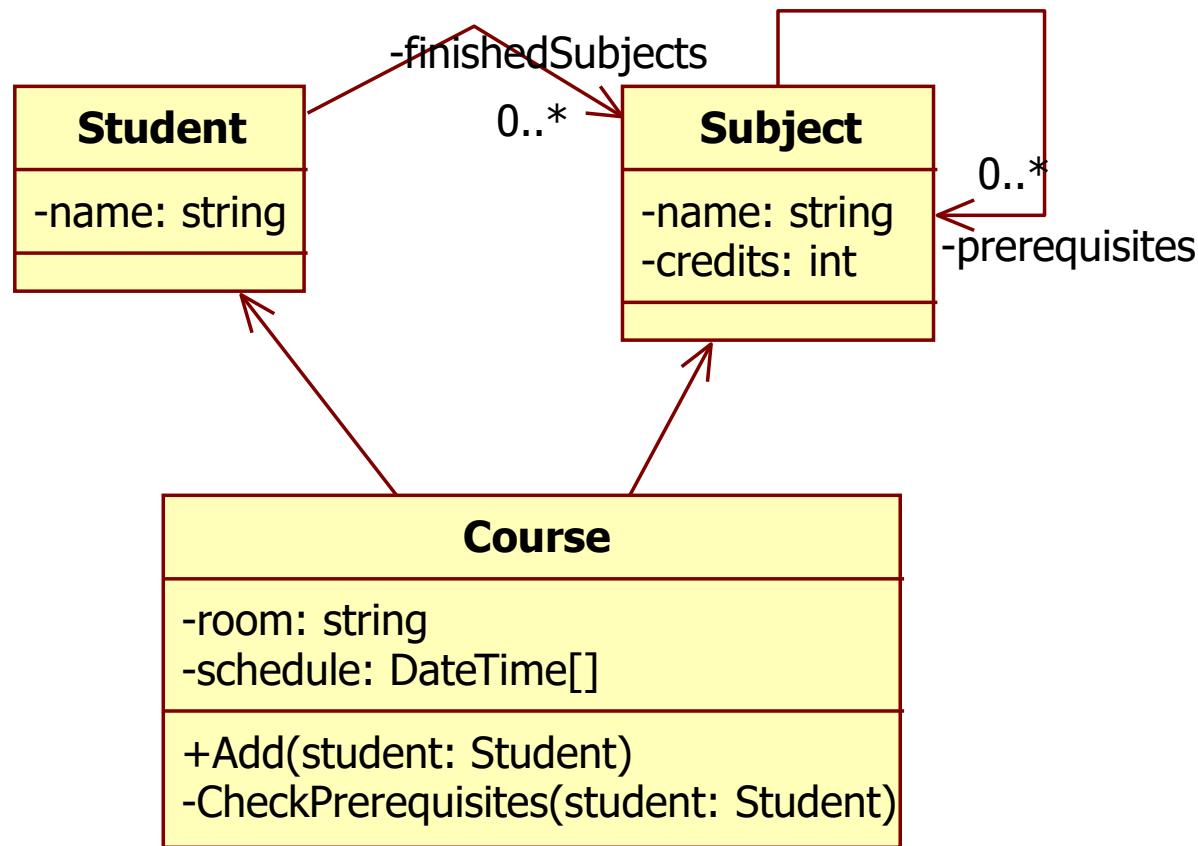
- Problems if violated:
 - some classes have too much responsibility, some have too little or none
 - if responsibility is not divided evenly, it usually shows as behavior god classes and classes with only accessor methods
 - usually unevenly distributed responsibility results from the wrong design process: procedural design instead of responsibility driven design

R1. Distribute responsibility horizontally evenly

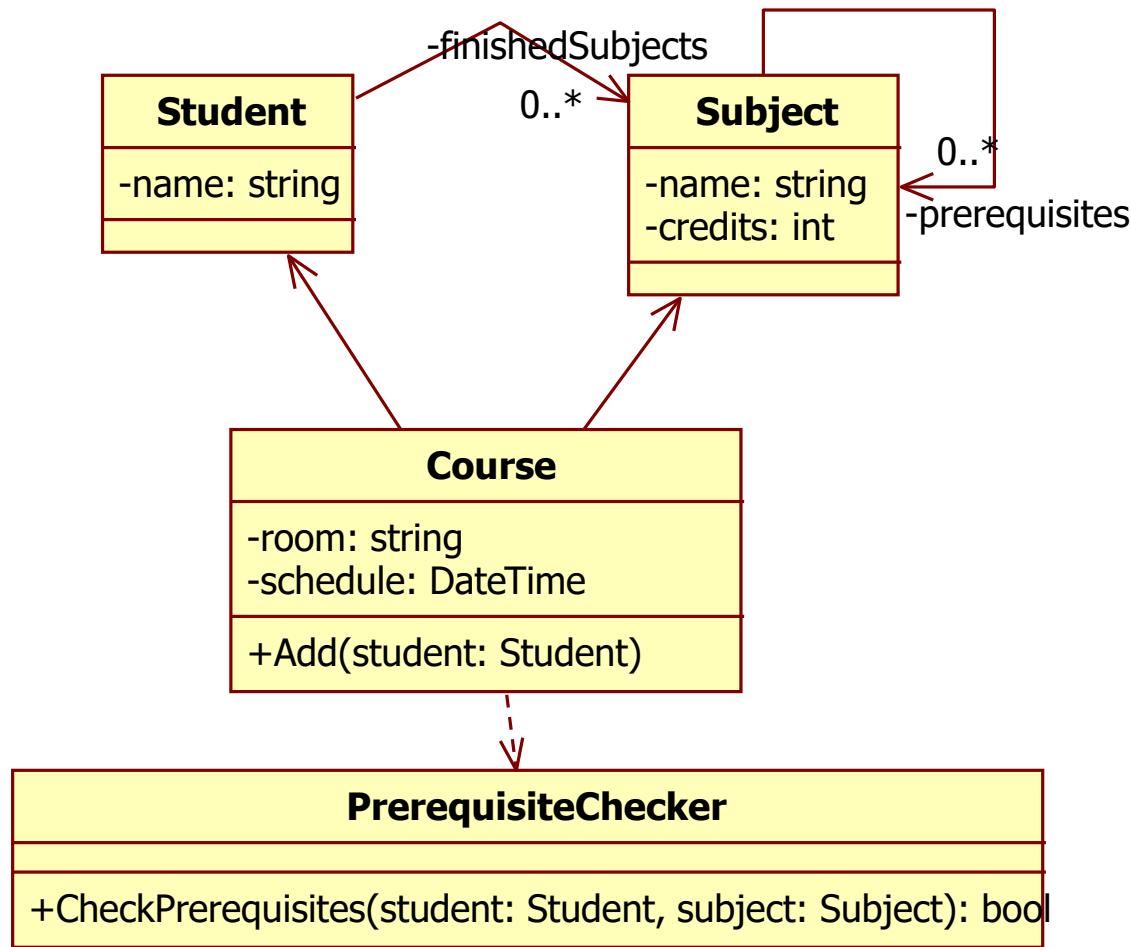
- Rules:

- do not design by identifying data members and operations
- use responsibility driven design (e.g. CRC cards)
- responsibilities will become operations
- data should only be considered after the responsibilities are assigned

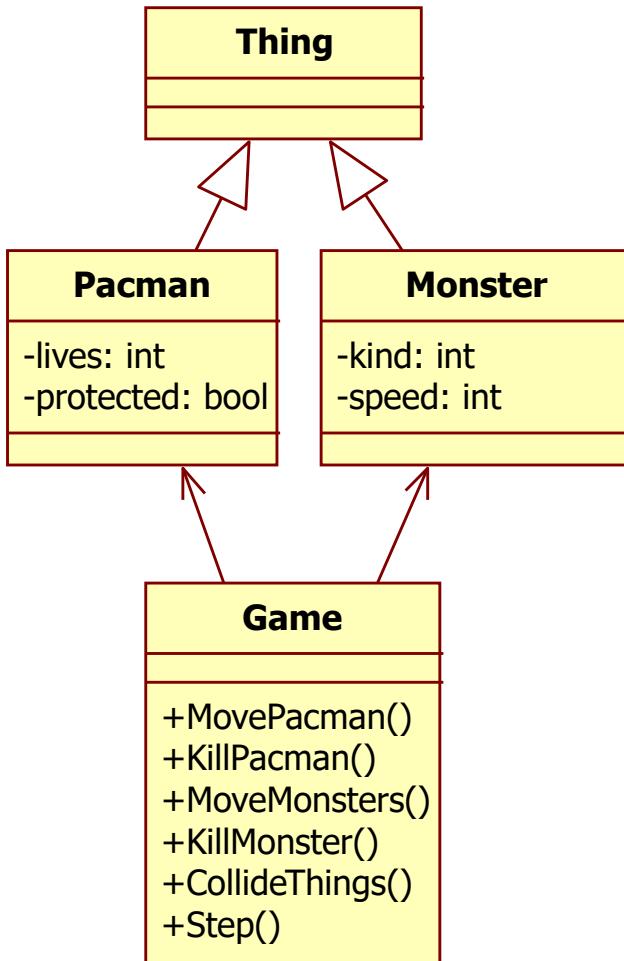
R1. Solution I.



R1. Solution II.



R2-R3. Problem



Problem: a god class + a lot of data classes

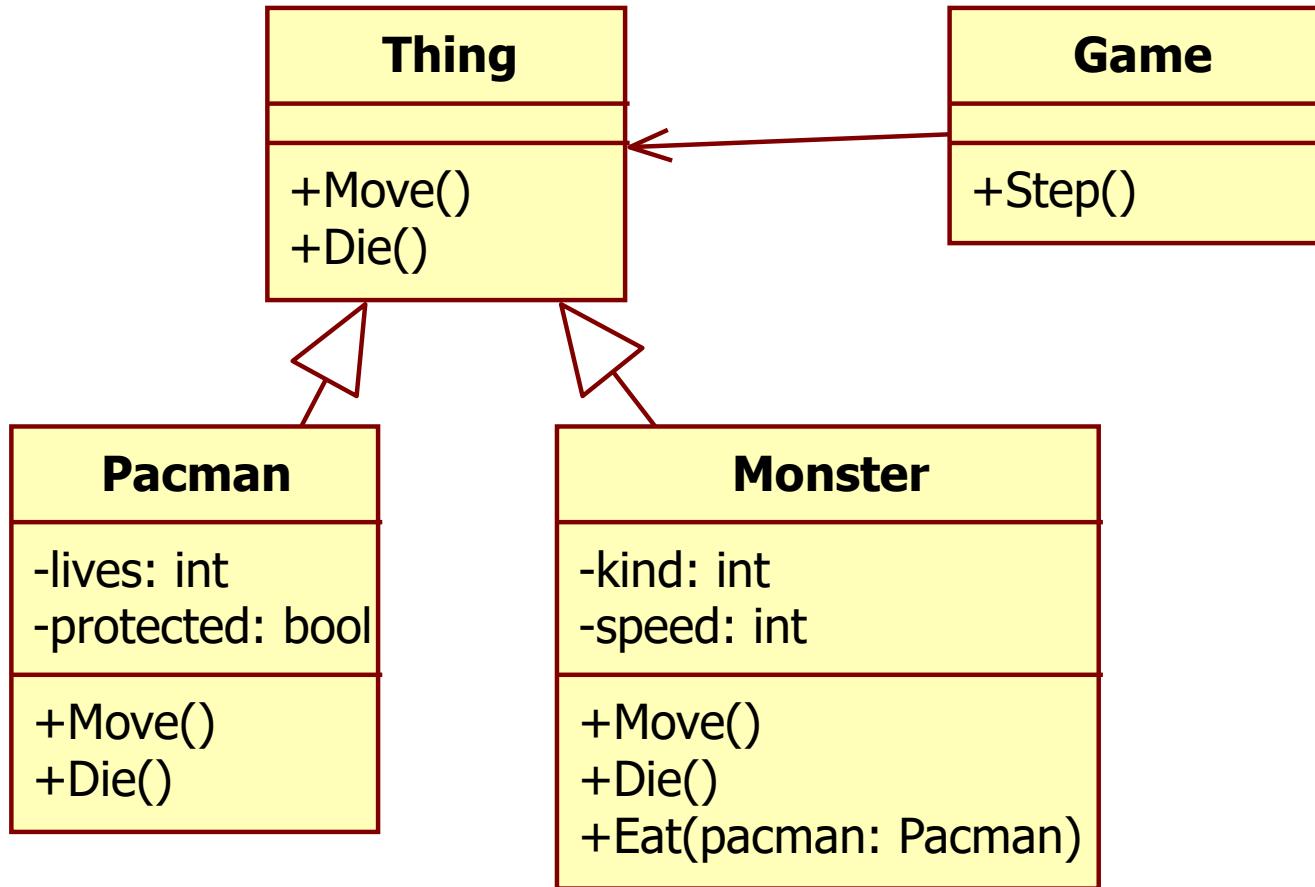
R2. Avoid god classes

- Problems if violated:
 - behavior is not at the right place
 - god classes have too much responsibility
 - violation of SRP
 - god classes are hard to change and maintain
- Rules:
 - avoid god classes
 - split up god classes and transfer the responsibilities to the appropriate classes
- Exceptions:
 - intentionally externalized behavior (but these are not god classes, either)
 - e.g. visitor design pattern, strategy design pattern, etc.

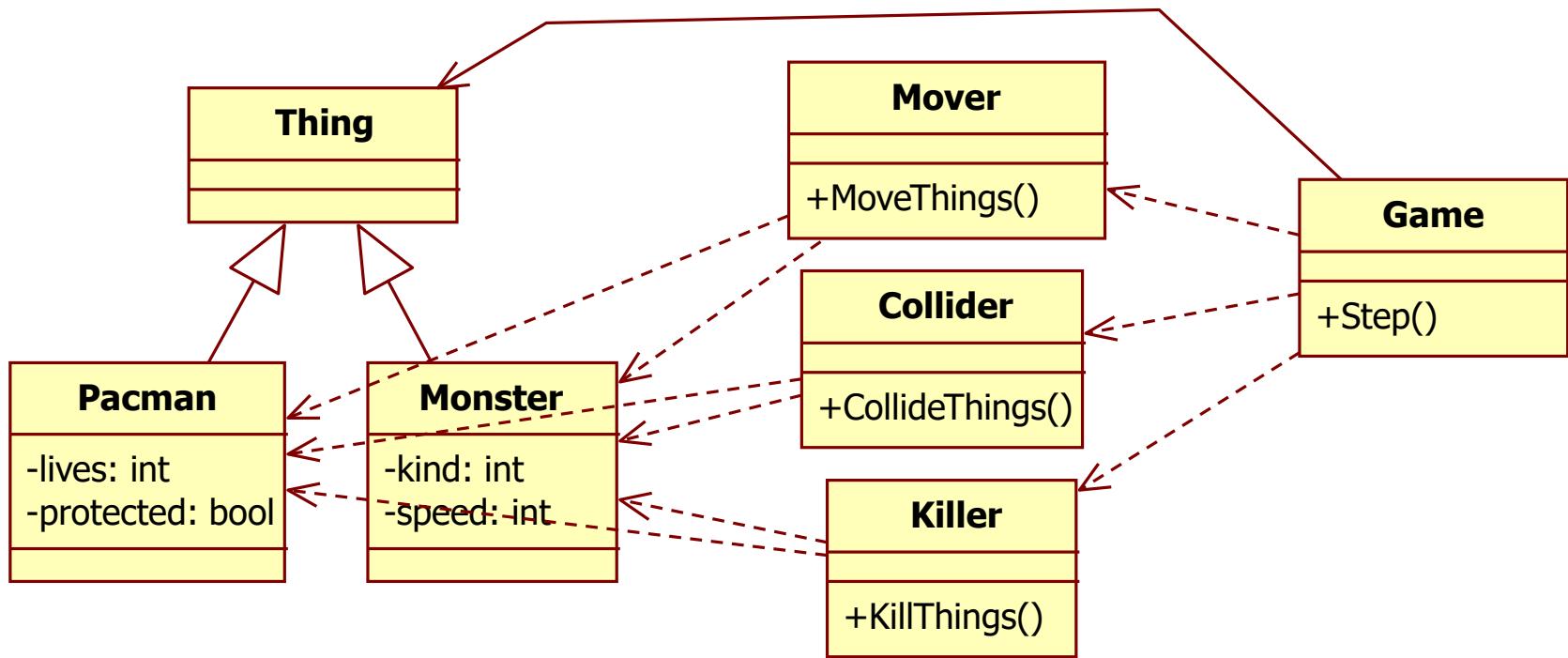
R3. Avoid classes with only accessor methods

- Problems if violated:
 - classes with only accessor methods have no behavior
 - behavior and data are not at the same place
 - the design can lead to god classes
- Rules:
 - avoid classes with only accessor methods
 - consider moving behavior from the users of this class into this class
 - e.g. decision made on information acquired through getters
- Exceptions:
 - ORM and DTO classes are OK

R2-R3. Solution



R4. Problem

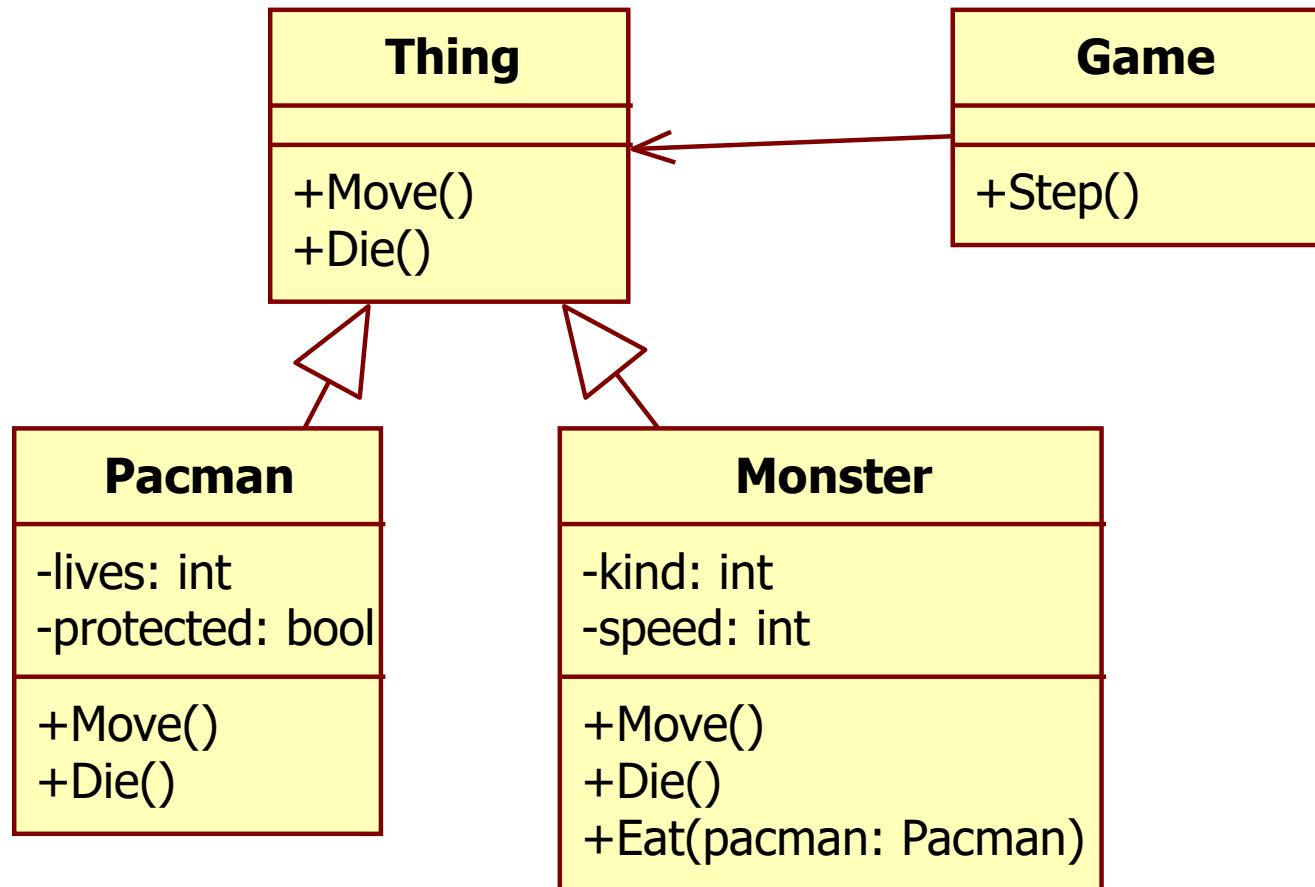


Problem: a lot of classes with a single method

R4. Avoid classes which should be methods

- Problems if violated:
 - behavior is not at the right place
 - cohesion of the class is broken
- Rules:
 - avoid classes which should be methods
 - be suspicious if the name of the class is a verb or derived from a verb
 - be suspicious if the class has only one meaningful behavior
 - consider moving the behavior to some existing or undiscovered class
- Exceptions:
 - intentionally detached behavior for improving reusability
 - e.g. visitor, strategy, etc.

R4. Solution



R5. Model the real world

- Problems if violated:
 - it may be hard to find the responsibilities of the classes
 - controller classes and agents divert responsibilities
- Rules:
 - model the real world whenever possible
 - the real world already works
 - during maintenance it is easier to find responsibilities
 - make sure you argue about design issues and not about class names (e.g. things don't do anything)
- Exceptions:
 - it is perfectly OK to violate this rule, this is just a guideline to assign responsibilities

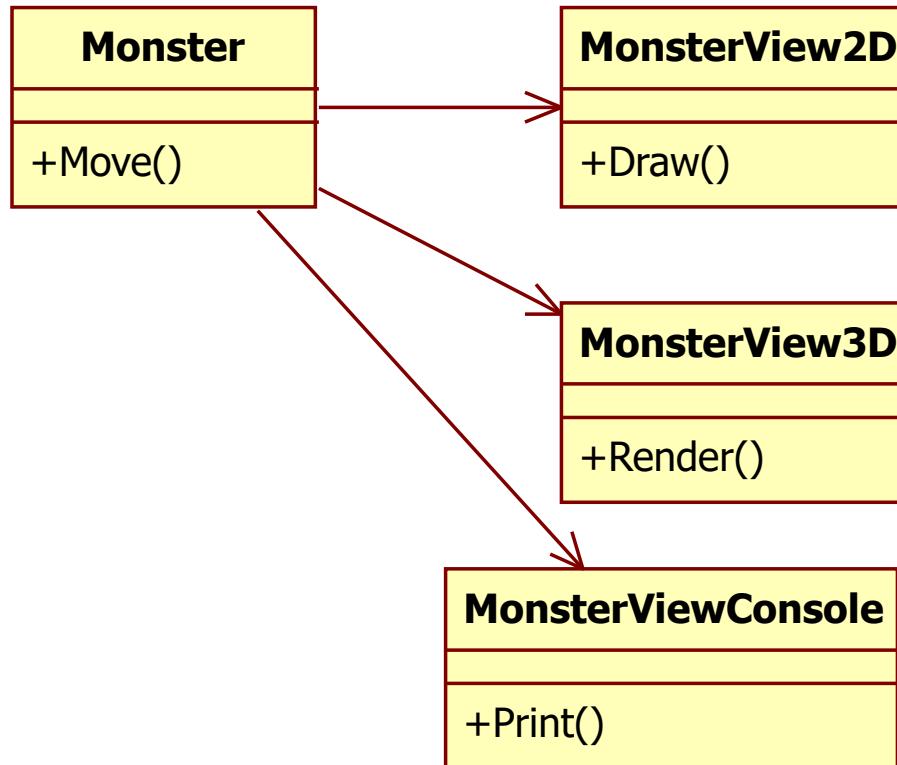
R6. Model at the appropriate abstraction level

- Problems if violated:
 - sticking to the real world may result in inflexible and unmaintainable models
- Rules:
 - introduce abstract agents in the design to decouple parts of the application from each other
 - for example, separating the domain from technology-specific parts (e.g. network)
 - introduce and keep abstract agents if they capture a useful abstraction
 - remove irrelevant real world agents, if they overly complicate the design
 - eliminate irrelevant classes with no meaningful behavior from the system

R7. Model only within the domain boundaries of the system

- Problems if violated:
 - parts outside the system are modeled, which is superfluous
 - maybe the boundaries of the system are not clear
 - e.g. user behavior is modeled
- Rules:
 - find the clear boundaries of the system
 - typically: user interaction, communication with external systems
 - be suspicious of classes which send messages to the system but receive no messages
 - do not model physical devices outside the system
 - do not model users and external systems, only provide an interface for them

R8. Problem



Problem: the model depends on the view

R8. The view should be dependent on the model and not the other way around

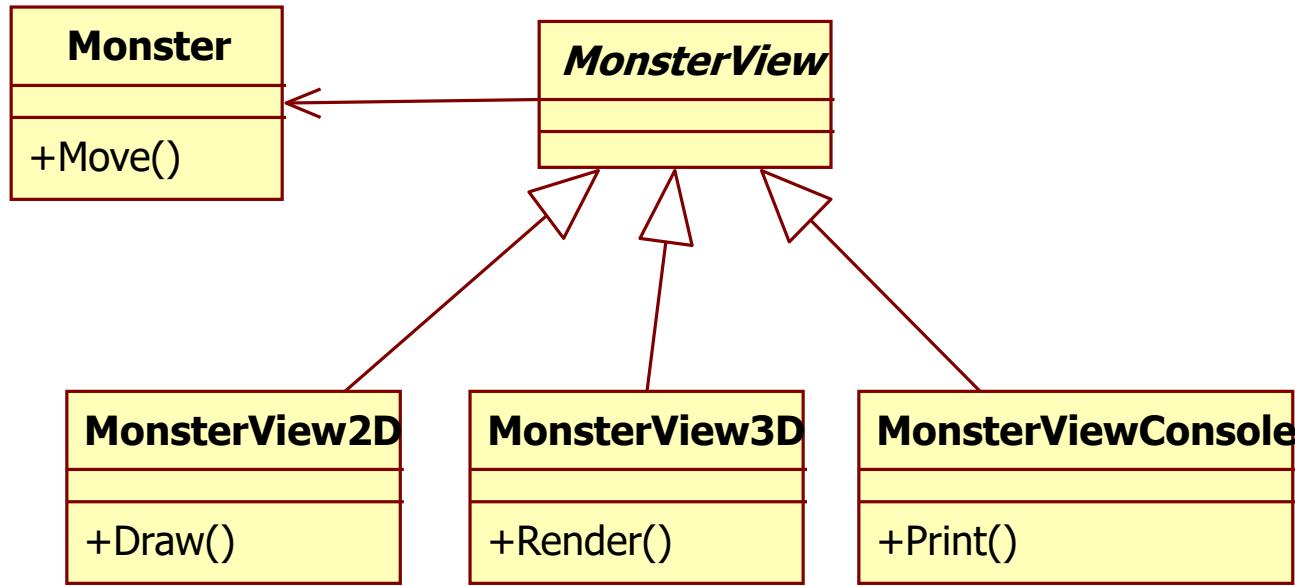
- Problems if violated:

- new external interfaces will cause changes in the model
 - e.g. new GUI, network communication, etc.
 - the view cannot be replaced without changing the model

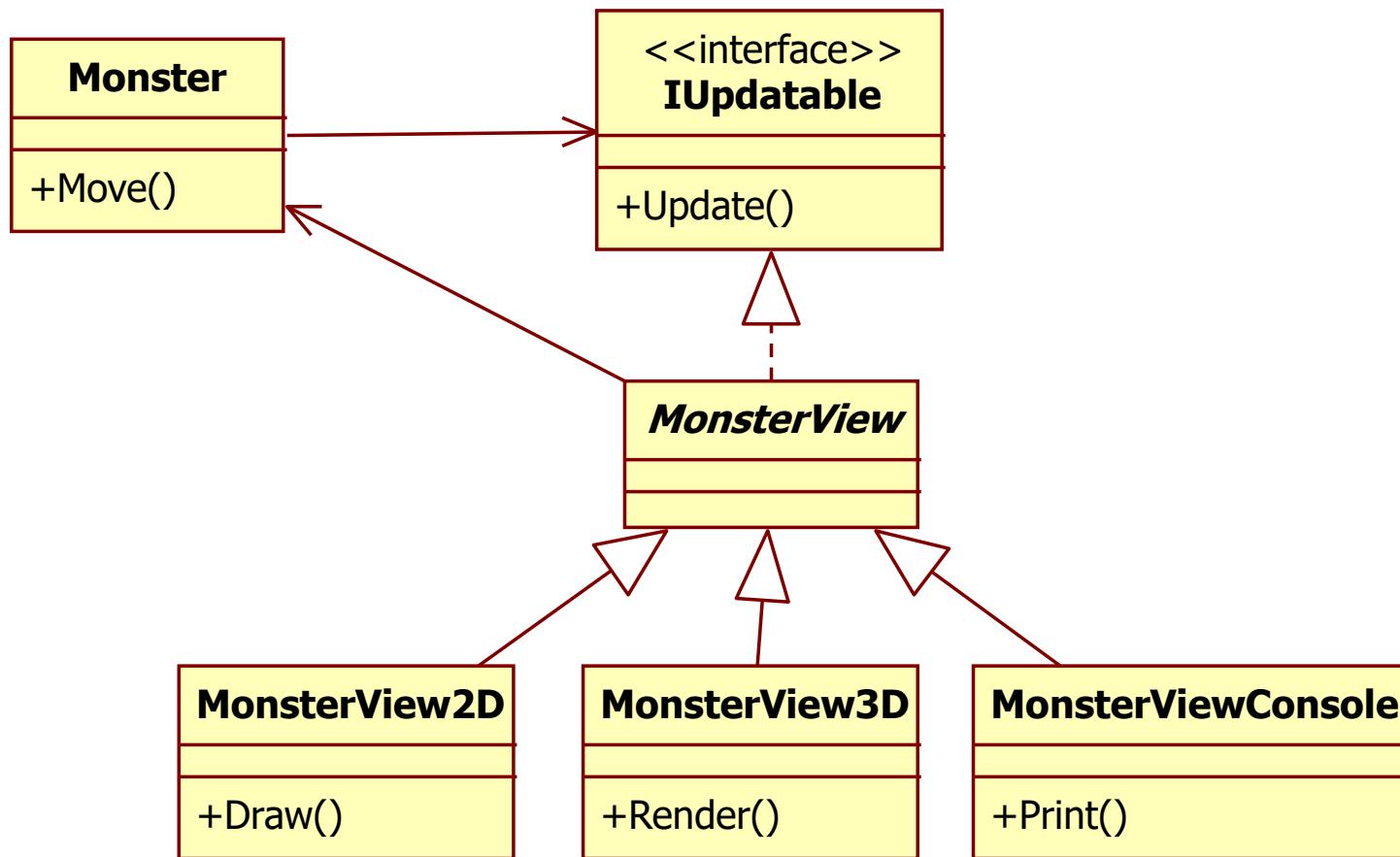
- Rules:

- the view should be dependent on the model
 - the model should be independent of the view and external communication
 - use DIP

R8. Solution I. – pull

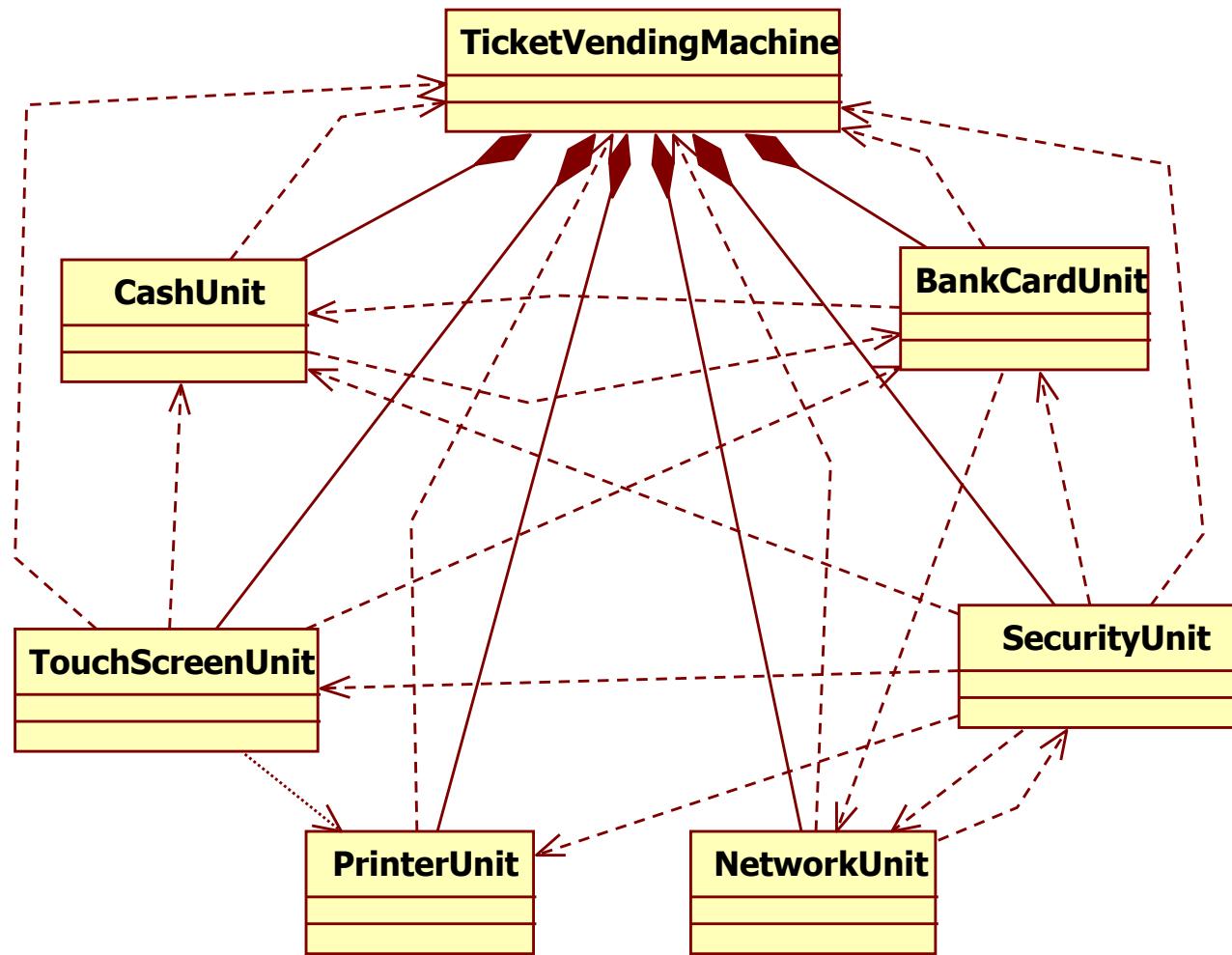


R8. Solution II. – push



Associations

A1-7. Problem



Problem: too many interdependencies

A1. Minimize the number of collaborating classes

- Problems if violated:
 - the class has many dependencies
 - the design is inflexible, it is hard to change
 - there is too much coupling between the classes
- Rules:
 - minimize the number of collaborating classes
 - there should be at most seven collaborators
 - so that the developer can keep them in his short term memory
 - use ISP and DIP to decrease dependency

A2. Minimize the number of different method calls between collaborating classes

- Problems if violated:

- more method calls mean more complexity and more coupling
 - it may also indicate the violation of DRY

- Rules:

- minimize the number of different method calls between collaborating classes
 - check whether the method calls violate DRY and if yes, introduce a new method for the task
 - check whether you have a god class with too much responsibility, and if yes, split it up
 - check whether ISP is not violated

A3. Distribute responsibilities in containment relationships

- Problems if violated:
 - the implementation of a class may become complex
 - parts of it cannot be reused
- Rule:
 - distribute responsibilities in deep and narrow containment hierarchies
 - the containers should be black boxes, the users of the class should not know about the internal structure
 - make sure not to violate LoD

A4. Prefer containment over association

- Problems if violated:
 - association is not black-box usage
 - the users of the class have to know about the associated objects
 - when they create an instance of the class they also need to pass the associated objects, so they depend on them, too
- Rules:
 - when given a choice, prefer containment over association
 - avoid circular containment
- Exception:
 - if containment cannot be used
 - if the associated objects have to be known by others

A5. A container object should use the contained objects

- Problems if violated:
 - the contained object may be useless
 - the contained object is returned to others to be used by them
 - this is a violation of LoD
- Rules:
 - containment implies uses relationship, too
 - the container object should use the contained objects
 - do not return the contained objects to others
 - the container object should forward calls to the contained objects (LoD)
 - see the Façade design pattern
- Exception:
 - collection classes

A6. A contained object should not use its container object

- Problems if violated:
 - the contained object cannot be reused outside its container
 - circular dependency between the container and the contained object
- Rules:
 - a contained object should not depend on its container object
 - use DIP or ISP to change the direction of the dependency

A7. Contained objects should not communicate with each other directly

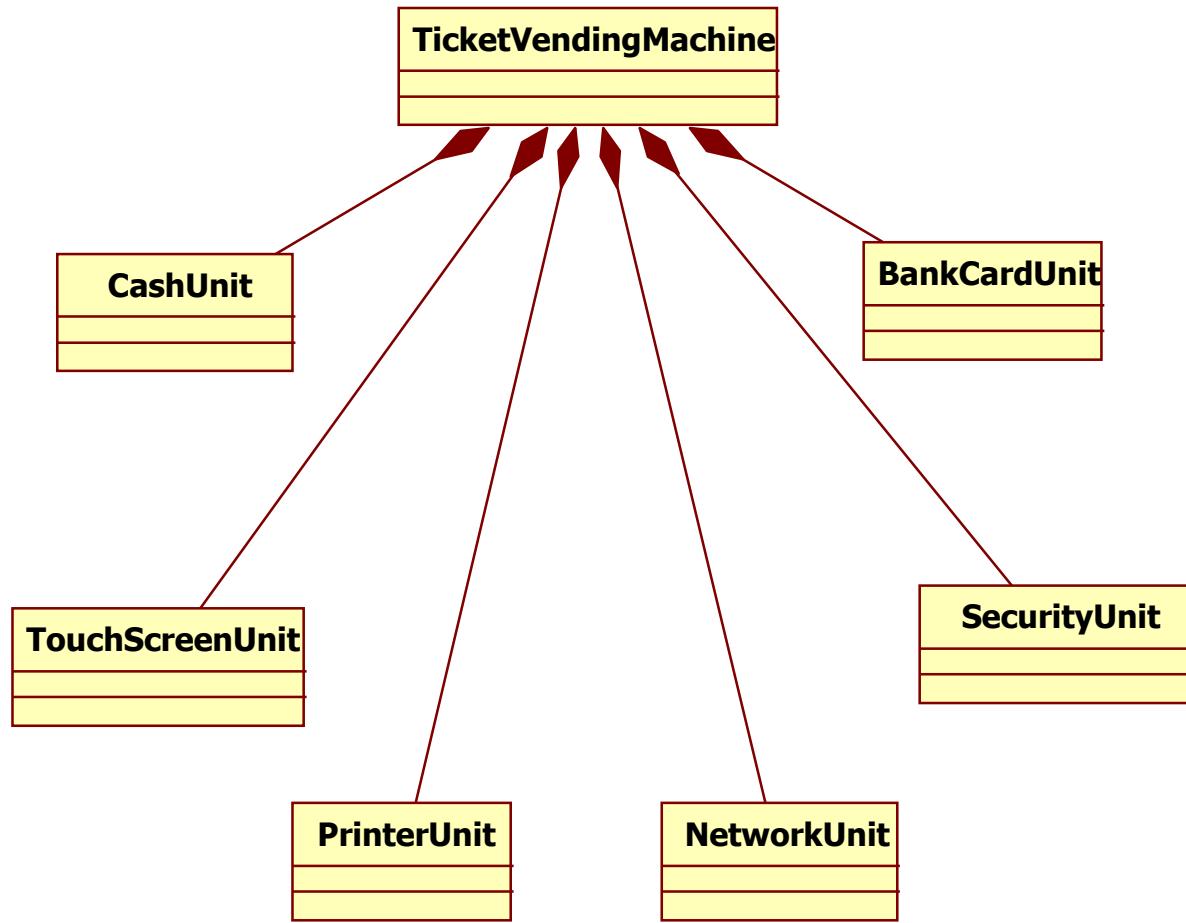
- Problems if violated:

- the container already knows the contained objects, and if they know each other, there is too much interdependency
 - the components cannot be reused without each other

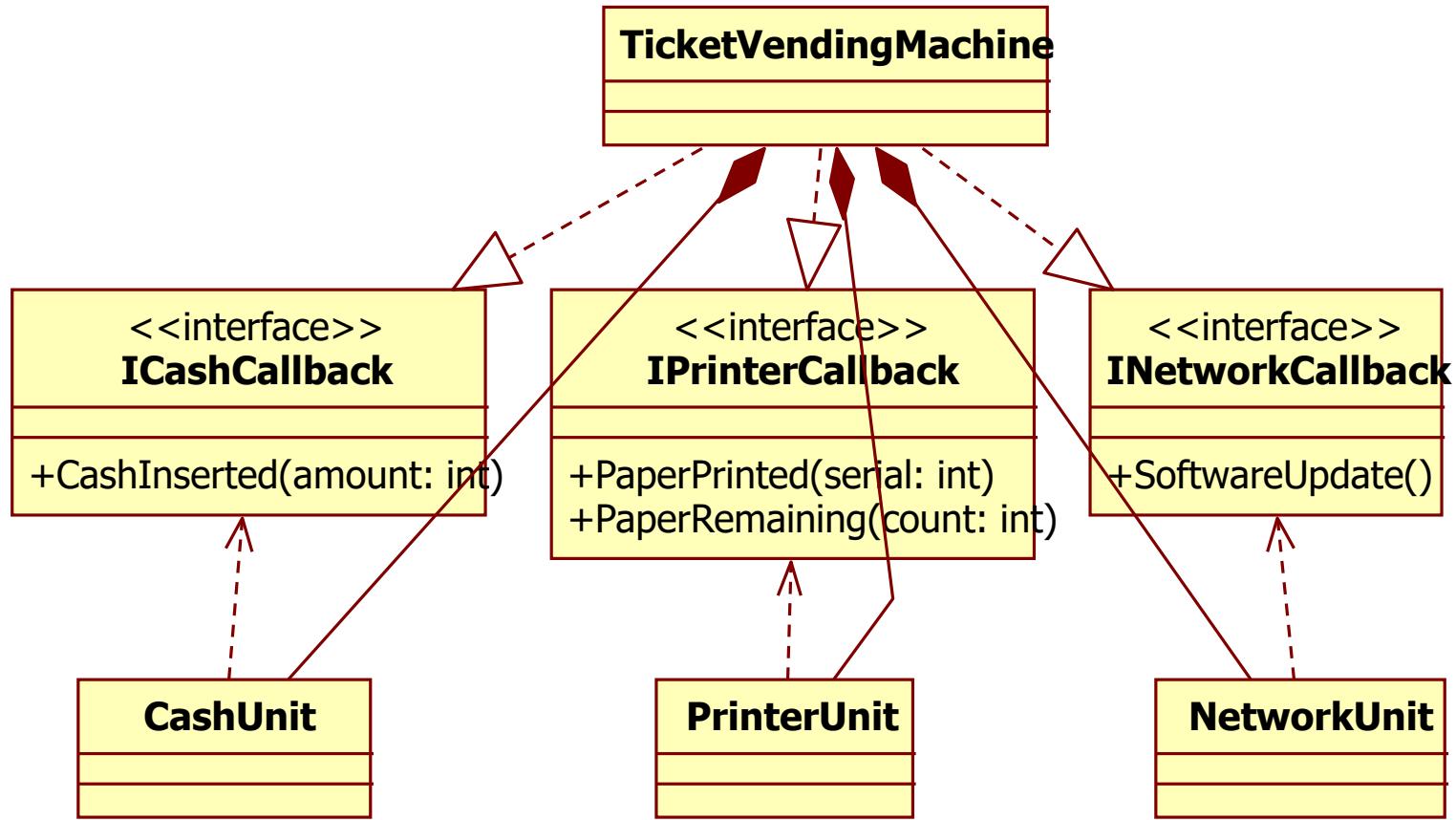
- Rules:

- contained objects in the same container should not communicate with each other directly
 - use the dependency of the container on the contained objects to perform the task
 - see also the Mediator design pattern
 - do not introduce new dependencies between the contained objects
 - better to have circular dependency between the container and contained objects ($2N$ dependencies) than heavy interdependency between contained objects (N^2 dependencies)
 - still better to use DIP or ISP with callback interfaces to remove circular dependency
 - see also the Observer design pattern (events in C#)

A1-7. Solution I.

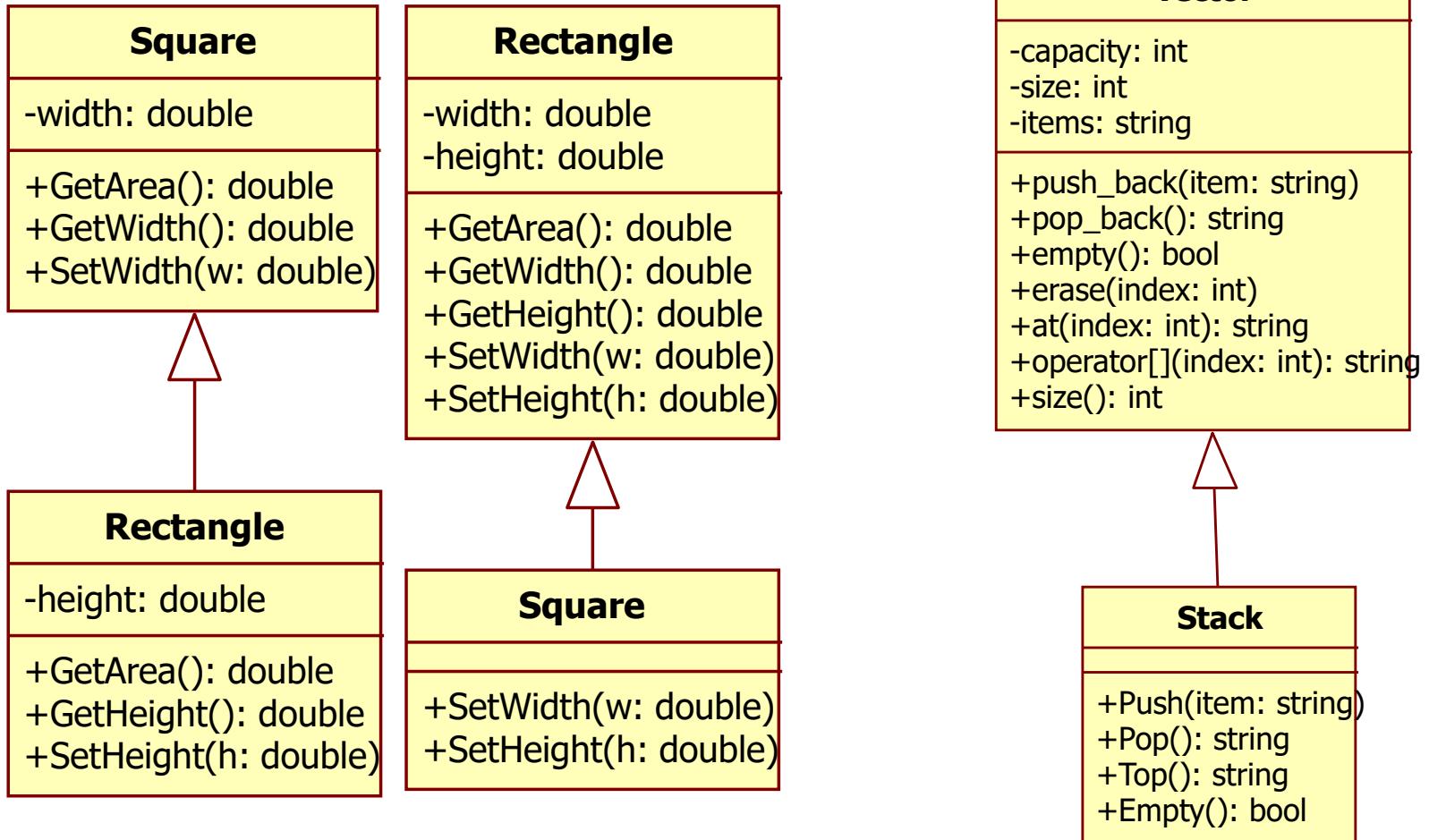


A1-7. Solution II.



Inheritance

I1-2. Problem

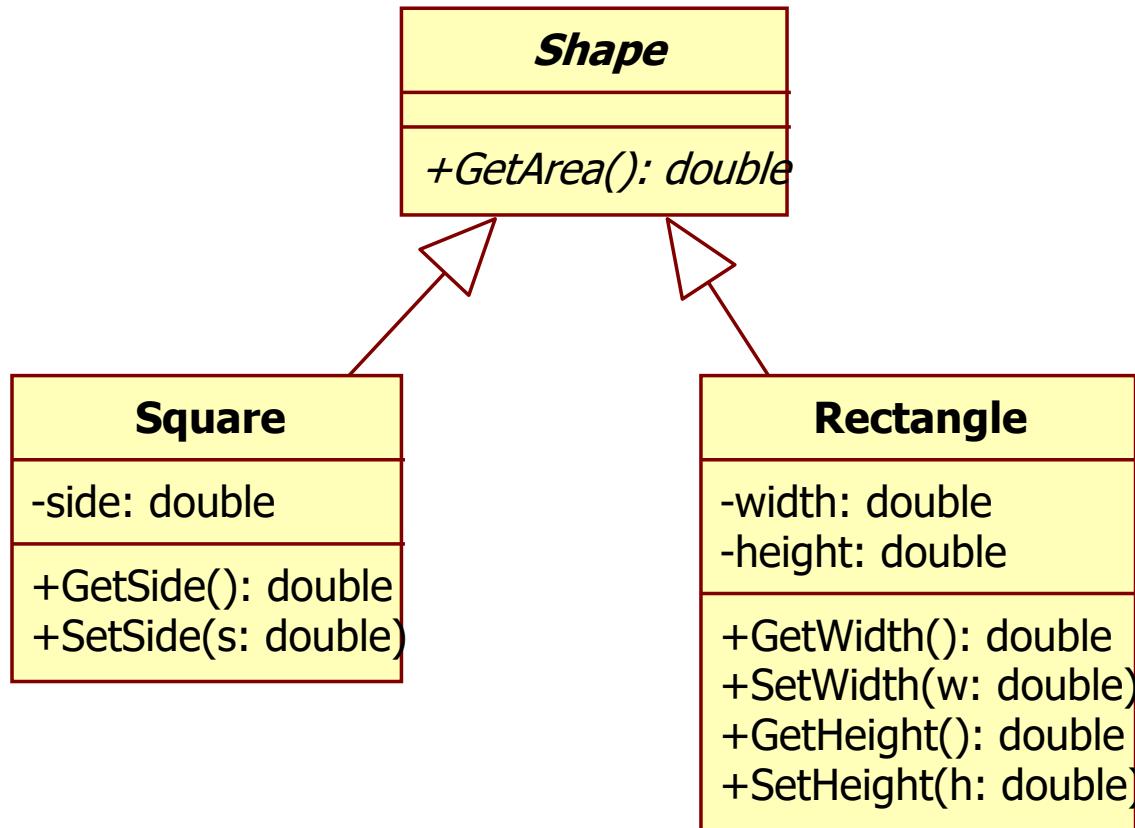


Problem: inheritance for data reuse

I1. Inheritance should always be behavior specialization

- Problems if violated:
 - it leads to the violation of LSP
 - similar to the square-rectangle problem
 - cannot hide the methods from the base class
- Rules:
 - inheritance should be about behavior reuse and specialization of behavior (LSP)
 - never use inheritance for data reuse
 - for data reuse use containment and delegation
 - e.g. implementing a stack using a list

I1. Solution



I2. Prefer containment over inheritance

- Problems if violated:

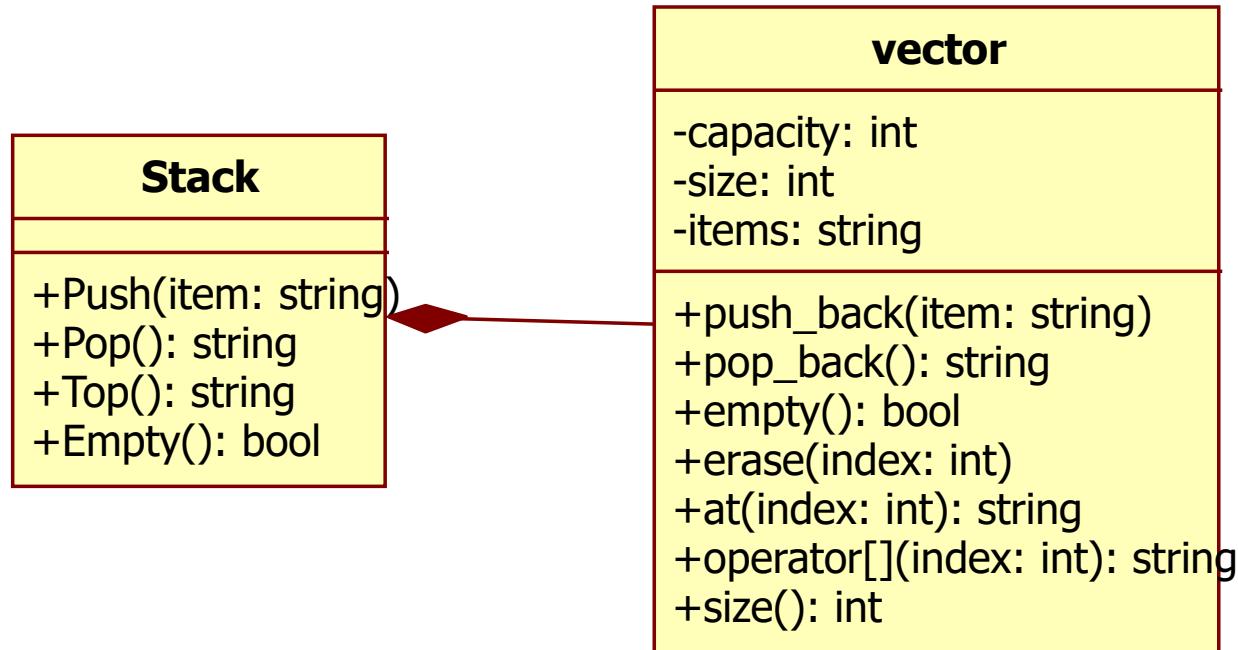
- inheritance is used for data reuse
- exposing too much implementation details
- public methods of the base class are also published
- violation of LSP
- inheritance is not black-box reuse
- hard to change implementation later

I2. Prefer containment over inheritance

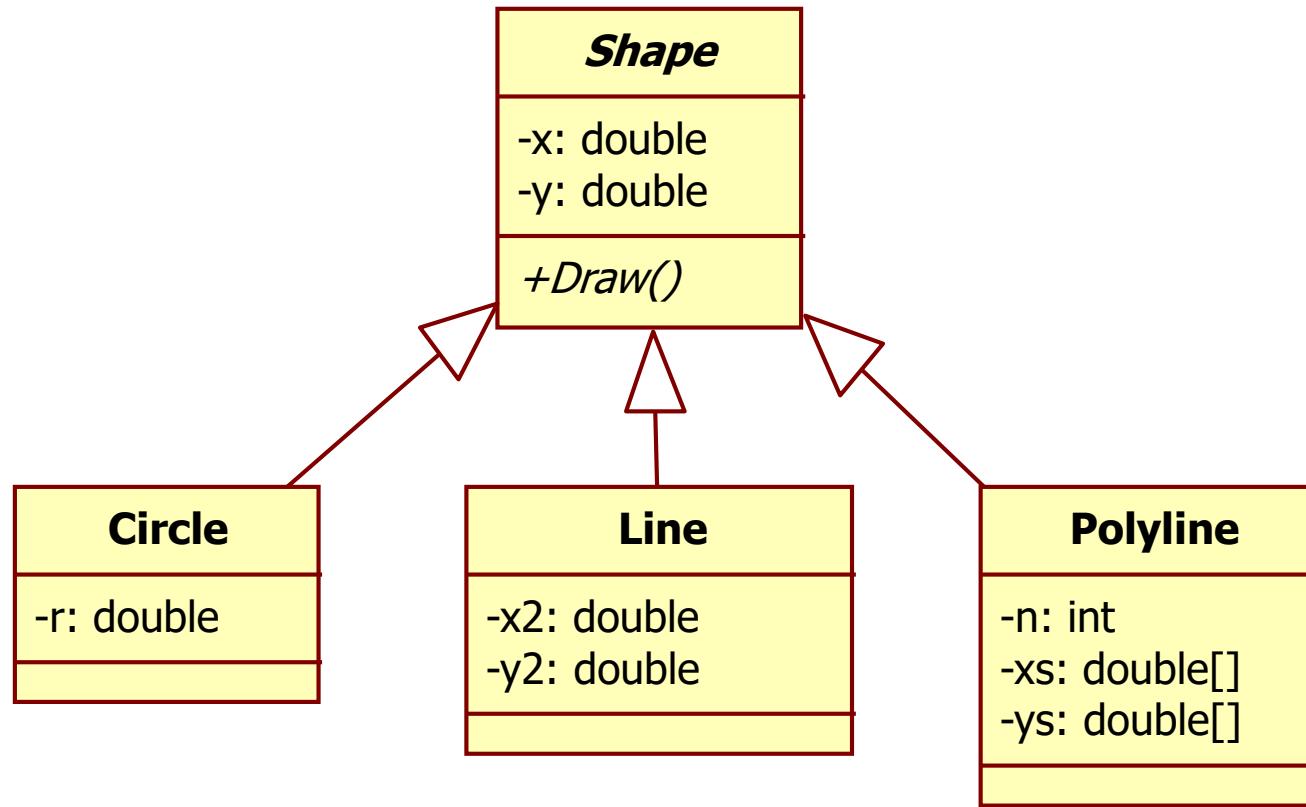
- Rules:

- if you can choose, prefer containment over inheritance
- if you are unsure, use containment
- never use inheritance for data reuse, always use containment in this case
- use inheritance only if the behavior is also reused
- ask the inheritance-containment questions
 - Is A a kind of B? Is A a part of B?
- don't violate LSP
- don't violate the contract of the base class
 - pre- and post-conditions, invariants
- behavior can also be added by using the Decorator design pattern
 - e.g. thread-safe collections, checking contracts, etc.

I2. Solution



I3. Problem

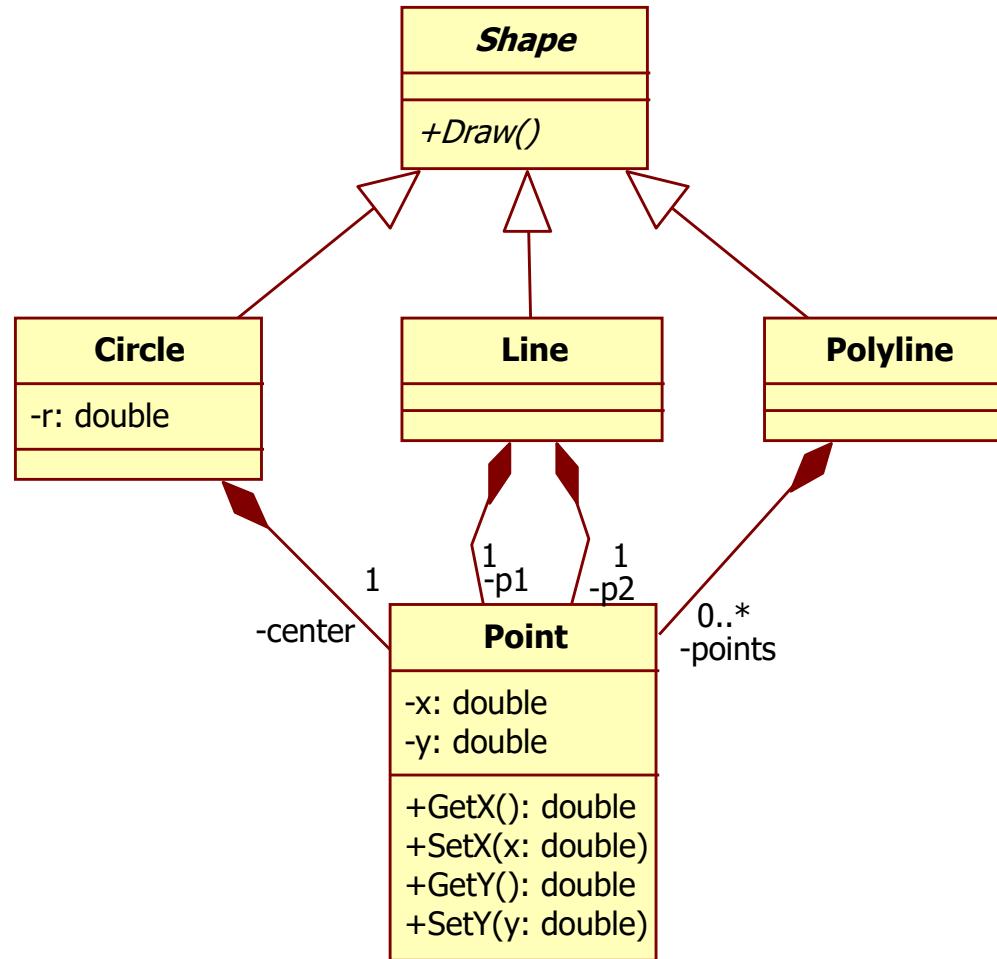


Problems: shared data without shared behavior, parallel arrays, special handling of data

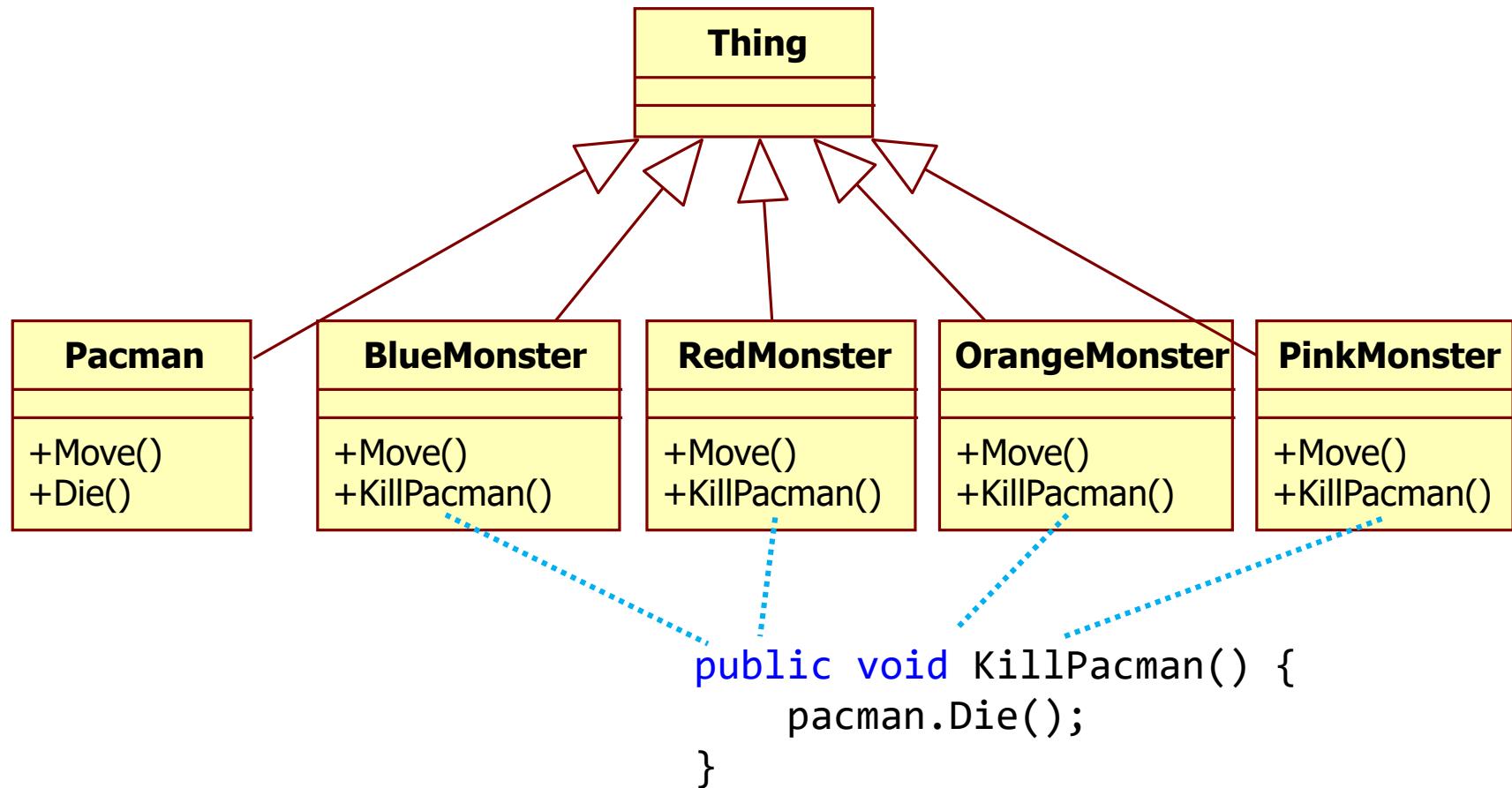
I3. Shared data without shared behavior should be in a containment relationship

- Problems if violated:
 - violation of LSP
- Rules:
 - if data is shared between two classes without shared behavior, then the data should be placed in a class that will be contained by each sharing class

I3. Solution



I4-5. Problem



Problem: shared behavior repeated

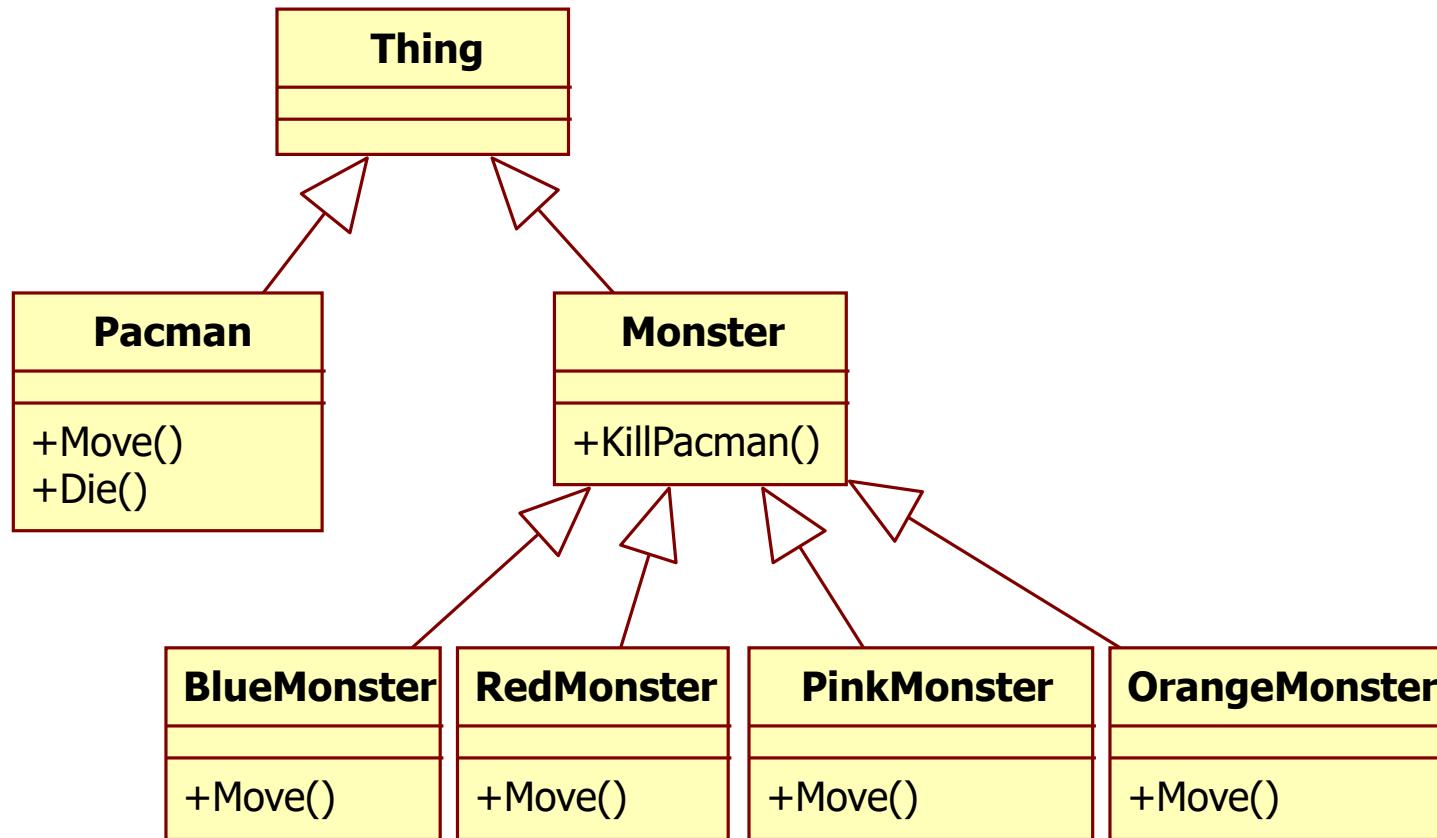
I4. Shared data with shared behavior should be in a common superclass

- Problems if violated:
 - the shared behavior has to be implemented in multiple places
 - this is a violation of DRY
- Rules:
 - shared behavior means a common abstraction
 - if two classes have common data and behavior, then those classes should each inherit from a common base class that captures those data and behavior
- Exception:
 - if multiple inheritance is required and the language does not support it
 - preventing the violation of DRY: use delegation, see the Strategy design pattern

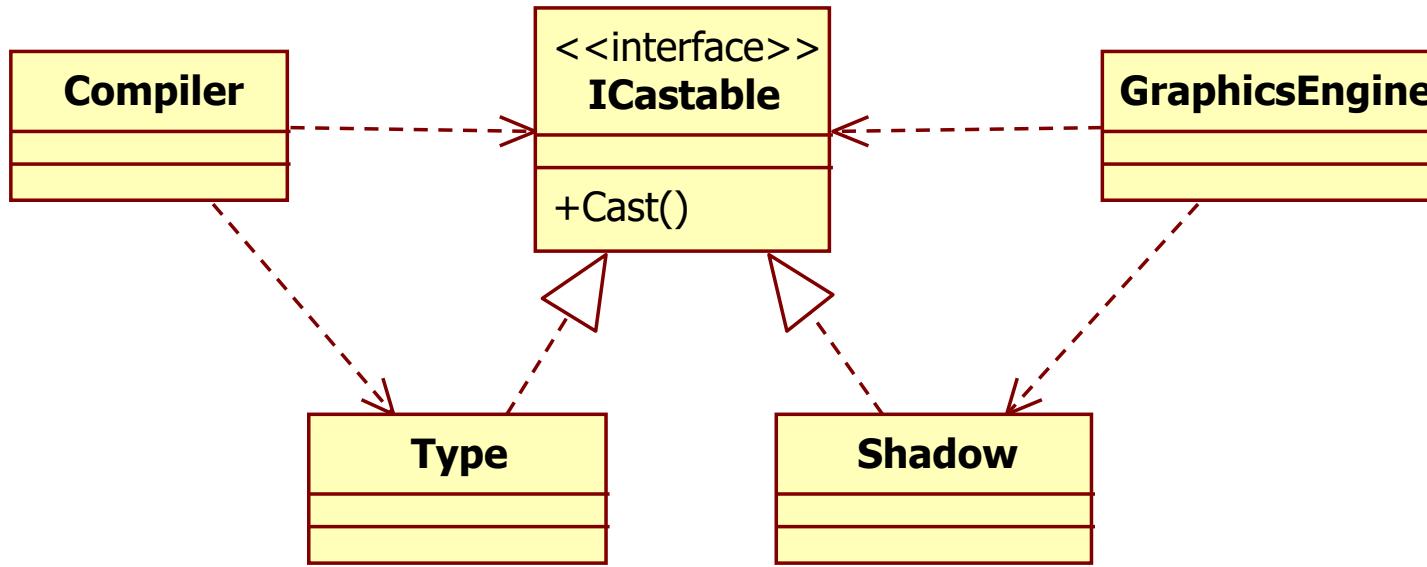
I5. Move common data and behavior as high as possible in the inheritance hierarchy

- Problems if violated:
 - the common data and behavior may appear in multiple places violating DRY
- Rules:
 - move common data and behavior as high as possible in the inheritance hierarchy
 - this way the descendants can take advantage of the common abstraction
 - this also allows users of a derived class to decouple themselves from that class in favor of a more general class

I4-5. Solution



I6. Problem



Problem: shared interface without similar behavior

I6. A common interface should only be inherited if the behavior is also shared

- Problems if violated:
 - violation of LSP: using an interface for an unintended purpose
- Rules:
 - a common interface should only be inherited if the descendants are used polymorphically, and they have similar behavior

Duck typing in C#

- Duck typing:

- *"If it walks like a duck and it quacks like a duck, then it must be a duck."*

The diagram illustrates duck typing in C# through a code example. It shows two classes, `Duck` and `Goose`, each with a `Talk()` method that prints a specific sound. A `Speak(dynamic d)` method is defined to call the `Talk()` method on its argument. Red arrows point from the class definitions to the text "No common ancestor or interface!" and from the `d.Talk()` call to the text "Still handled the same way".

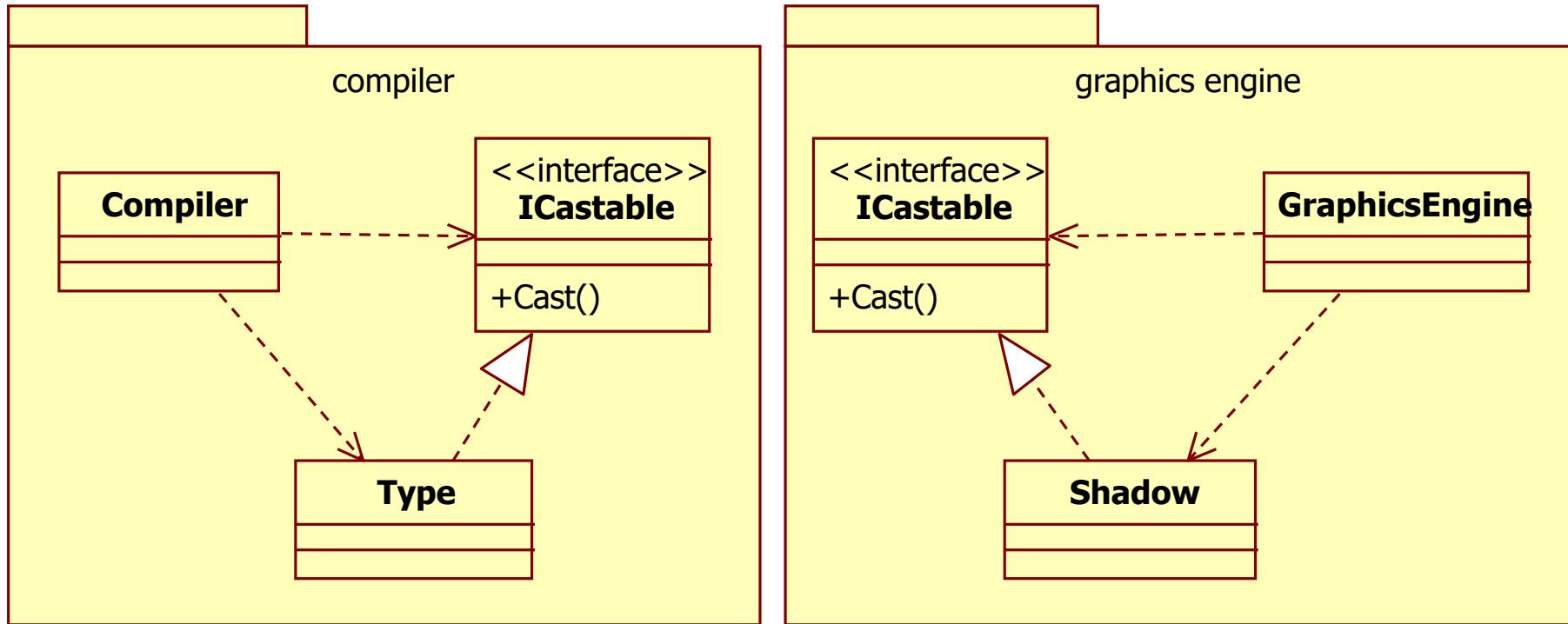
```
public class Duck {  
    public void Talk() { Console.WriteLine("Quack!"); }  
}  
public class Goose {  
    public void Talk() { Console.WriteLine("Honk!"); }  
}  
  
static void Speak(dynamic d) {  
    d.Talk();  
}  
  
dynamic bird1 = new Duck();  
dynamic bird2 = new Goose();  
Speak(bird1);  
Speak(bird2);
```

I6. A common interface should only be inherited if the behavior is also shared

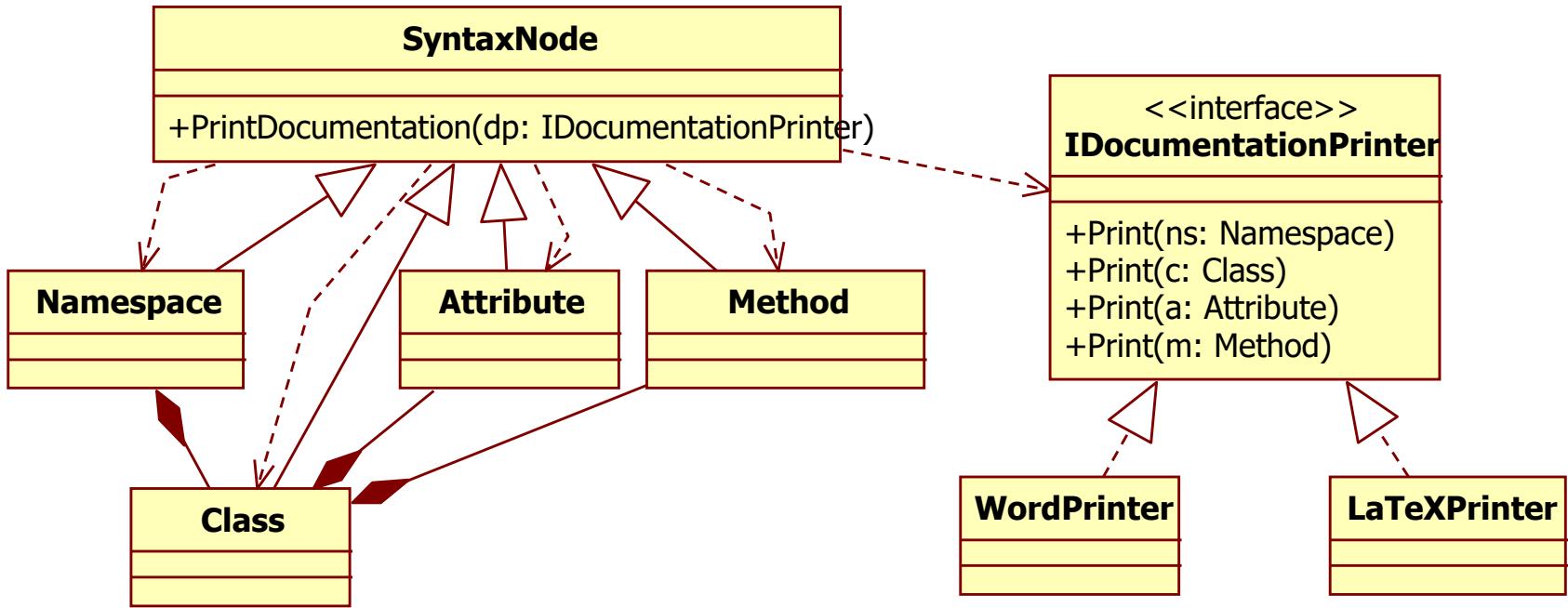
■ Rules:

- a common interface should only be inherited if the descendants are used polymorphically, and they have similar behavior
- beware of duck typing and be careful with C++ templates
 - duck typing: different types with the same set of methods treated as if they were the same type
 - strongly typed languages (e.g. Java) don't allow duck typing
 - duck typing is common in dynamic languages (e.g. JavaScript)
 - duck typing is also possible in C#: dynamic keyword
 - C++ templates are also a kind of implementation of duck typing

I6. Solution



I7. Problem



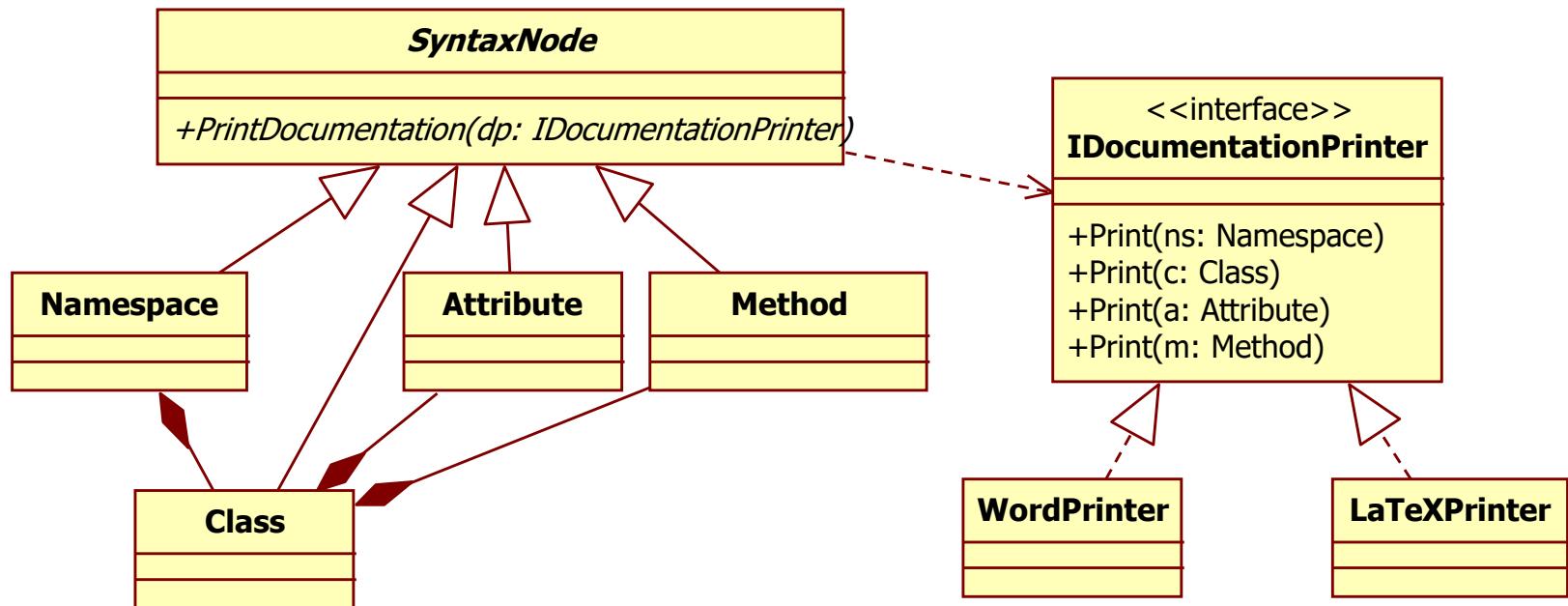
```
public void PrintDocumentation(IDocumentPrinter dp) {  
    if (this is Namespace) dp.Print((Namespace)this);  
    if (this is Class) dp.Print((Class)this);  
    // ...  
}
```

Problem: class depends on its descendants

I7. A class should not depend on its descendants

- Problems if violated:
 - if a class depends on its descendants, it will have to depend on its descendants added later
 - this is a violation of OCP
- Rules:
 - base classes should not depend on their descendants
 - descendants always depend on their base classes: inheritance is already a kind of dependency
- Note: not only `SyntaxNode.PrintDocumentation()` violates OCP
 - `IDocumentationPrinter` and its descendants also violate OCP, although for a different reason: when new nodes are added, new `Print()` functions are needed (see Visitor design pattern)
 - but at least the violation of OCP can be recognized at compile time
 - however, for `SyntaxNode.PrintDocumentation()` the violation of OCP cannot be recognized at compile time

I7. Solution



```
public class Namespace : SyntaxNode {  
    public void PrintDocumentation(IDocumentPrinter dp) {  
        dp.Print(this);  
    }  
}  
  
public class Class : SyntaxNode {  
    public void PrintDocumentation(IDocumentPrinter dp) {  
        dp.Print(this);  
    }  
}  
// ...
```

I8. Use protected only for methods and never for attributes

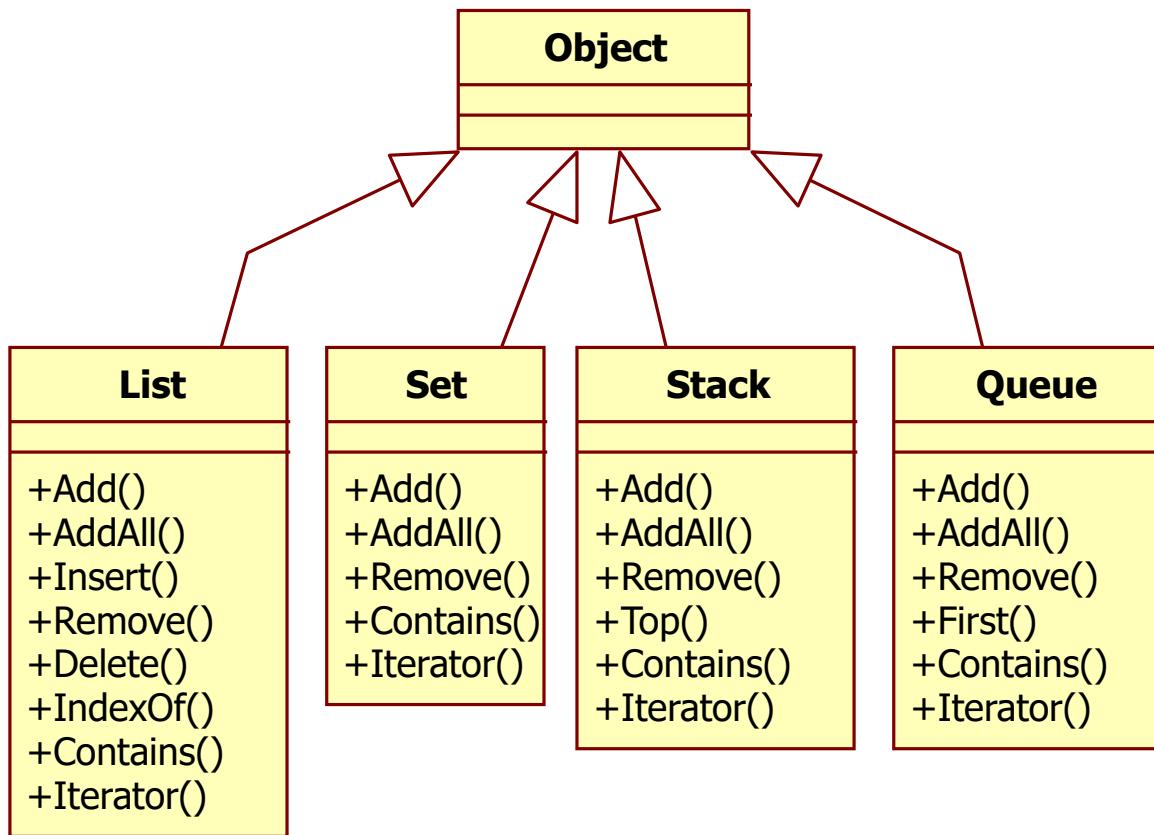
- Problems if violated:
 - descendants accessing protected attributes can bring the object into a bad state
- Rules:
 - attributes should always be private
 - if the value of attributes is needed, provide protected accessor methods for them
 - do not make these method public, if it is not necessary
 - they would reveal too much implementation detail
 - the users of the class may not need these
 - use template methods calling protected methods

Protected visibility

- Protected in C# and Java means that the member is available from a descendant class, but only through that or a more descendant class!
- Example:

```
class Monster {  
    protected Direction Direction { get; protected set; }  
    protected virtual void Move(Direction d) {  
        // ...  
    }  
    public virtual void Step() {  
        this.Move(this.Direction);  
    }  
}  
class BlueMonster : Monster {  
    public void Meet(Monster m) {  
        m.Move(this.Direction); // Cannot access protected  
                               // member via qualifier of type 'Monster'  
    }  
}  
class RedMonster : Monster {  
    public void Meet(RedMonster m) {  
        m.Move(this.Direction); // OK  
    }  
}
```

I9-11. Problem



Problem: what is the signature of AddAll()?

I9. The inheritance hierarchy should be deep, but at most seven levels deep

- Problems if violated:
 - shallow inheritance hierarchy means poor taxonomy and poor reuse
 - too deep inheritance hierarchies are hard to keep in mind
- Rules:
 - create deep inheritance hierarchies to provide a fine taxonomy and refined abstractions
 - keep the levels of inheritance at most seven so that the hierarchy can be kept in mind by developers

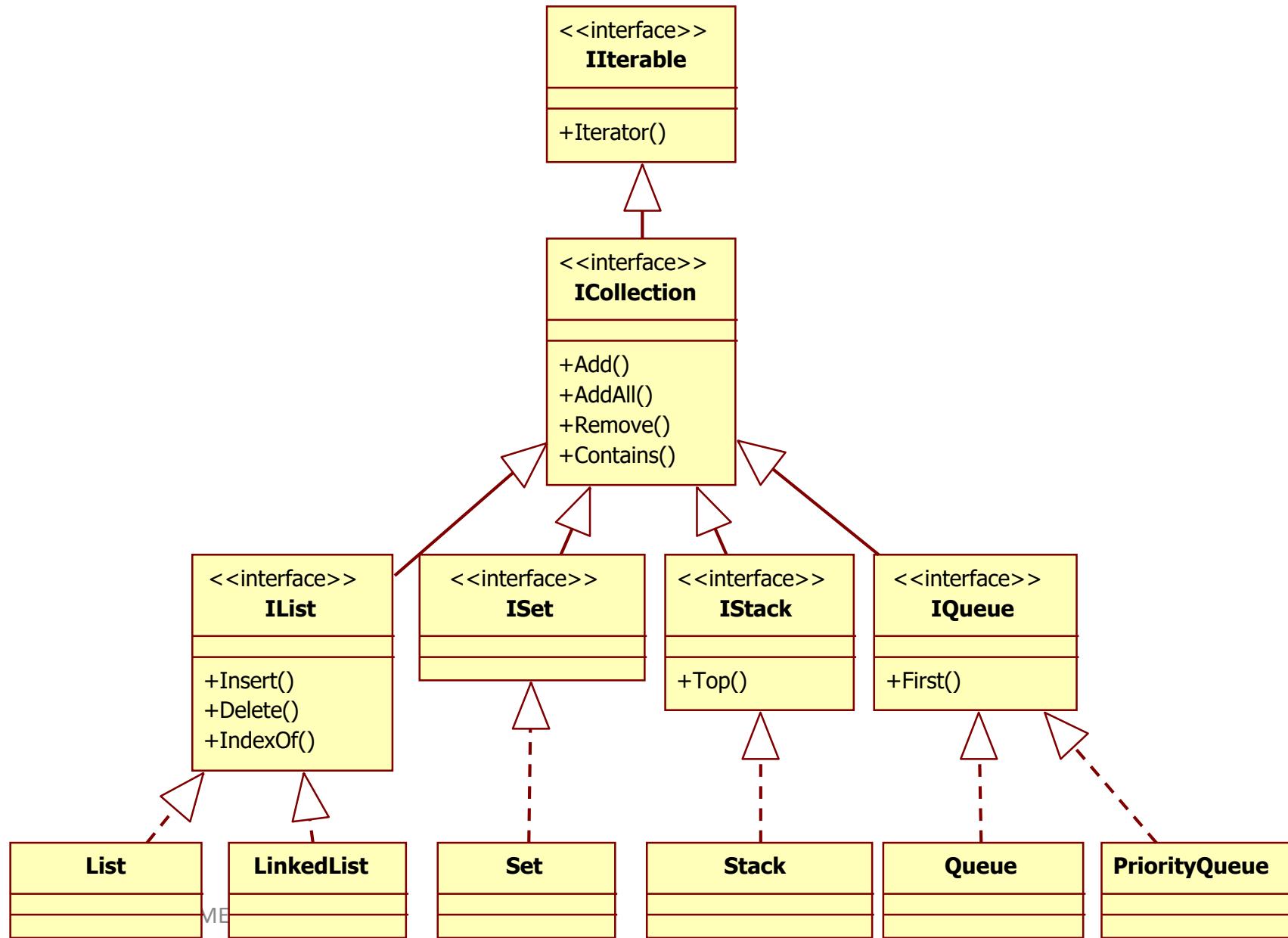
I10. Abstract classes should be at the root of the inheritance hierarchy

- Problems if violated:
 - abstract classes which are leaves in the inheritance hierarchy cannot be instantiated, so they are useless
- Rules:
 - abstract classes and interfaces should be at the root of the inheritance hierarchy
 - they can also appear in the middle, if they have further descendants
 - this rule is also a result of DIP, ISP and SDP
- Exception:
 - when writing a library to be used by others
 - e.g. defining expected interfaces, callback interfaces

I11. Roots of the inheritance hierarchy should be interfaces or abstract classes

- Problems if violated:
 - if a concrete class as a root is used throughout the system, then making it abstract later will be painful
- Rules:
 - roots of the inheritance hierarchy should be interfaces or abstract classes
 - especially between layers of the application
 - DIP, ISP and SDP also promote this rule
- Exceptions:
 - if the concrete class is not going to change, then it is perfectly OK to have it in the root
 - e.g. Thread class in Java

I9-11. Solution



I12. Problem

```
public class Pacman {  
    public void collide(Monster m) {  
        if (m is BlueMonster) {  
            // ... blue behavior  
        }  
        if (m is RedMonster) {  
            // ... red behavior  
        }  
        if (m is PinkMonster) {  
            // ... pink behavior  
        }  
        if (m is OrangeMonster) {  
            // ... orange behavior  
        }  
    }  
}
```

Problem: dynamic type checking

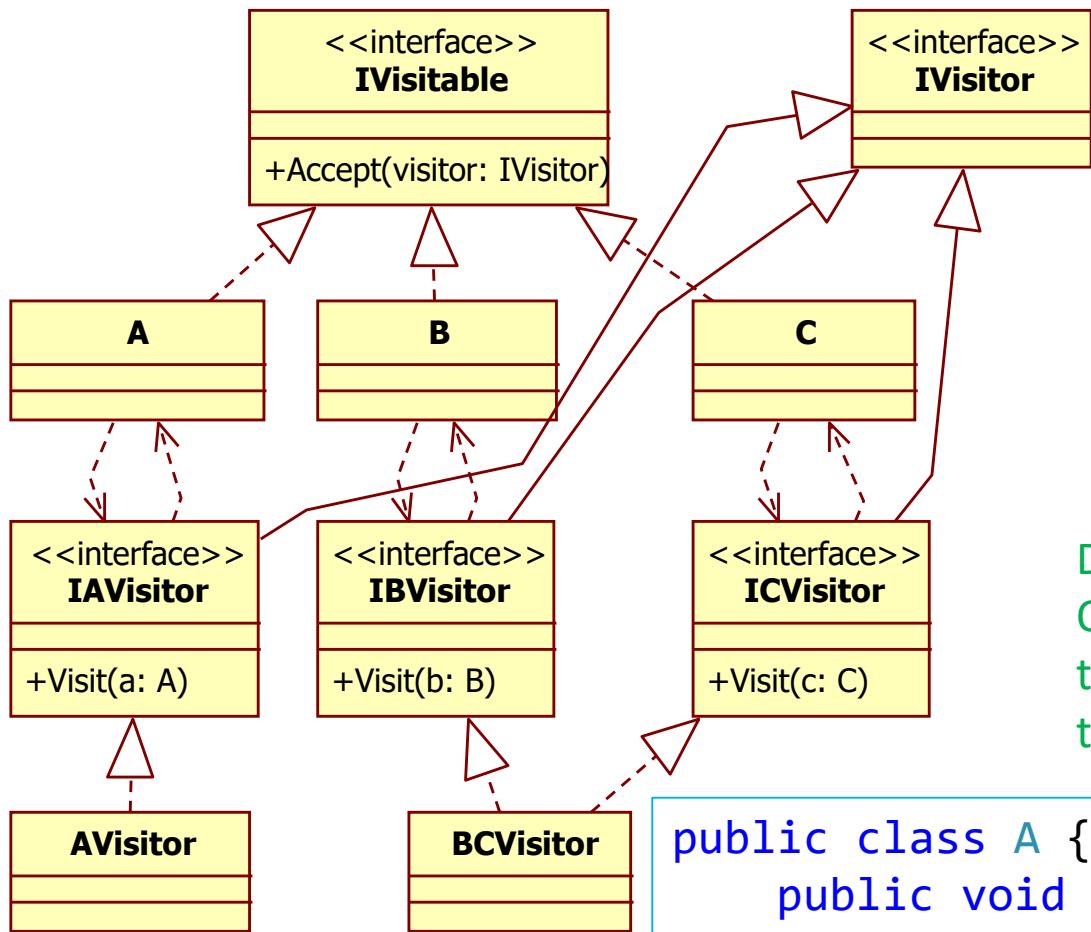
I12. Never test for the type of an object, use polymorphism instead

- Problems if violated:
 - responsibility is at the wrong place
 - violation of LSP and OCP
- Rules:
 - never test for the type of an object
 - introduce a method in the base class
 - override it in the descendants
- Exceptions:
 - sometimes tests for an interface are OK
 - e.g. Acyclic visitor design pattern, configuration

I12. Solution

```
public class Pacman {  
    public void Collide(Monster m) {  
        m.Eat(this);  
    }  
}  
public class BlueMonster {  
    public void Eat(Pacman p) {  
        // ... blue behavior  
    }  
}  
public class RedMonster {  
    public void Eat(Pacman p) {  
        // ... red behavior  
    }  
}  
// ...
```

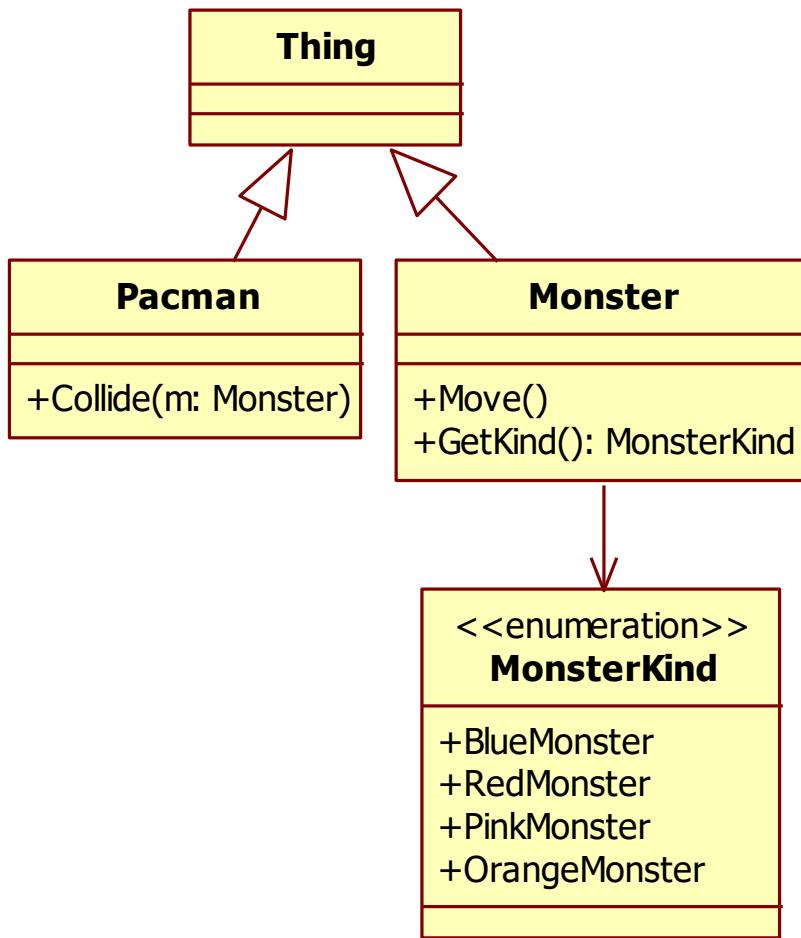
I12. Solution for a non-OCP-violating visitor: Acyclic Visitor



Dynamic type check is OK:
OCP and LSP are not violated, neither in
the IVisitable nor in the IVisitor part of
the inheritance hierarchy

```
public class A {  
    public void Accept(IVisitor visitor) {  
        if (visitor is IAVisitor) {  
            ((IAVisitor)visitor).Visit(this);  
        }  
    }  
}
```

I13. Problem



```
public class Pacman : Thing {
    public void Collide(Monster m) {
        switch (m.GetKind()) {
            case BlueMonster:
                // ... blue behavior
                break;
            case RedMonster:
                // ... red behavior
                break;
        }
    }
}

public class Monster : Thing {
    public void Move() {
        switch (kind) {
            case BlueMonster:
                // ... blue behavior
                break;
            case RedMonster:
                // ... red behavior
                break;
        }
    }
}
```

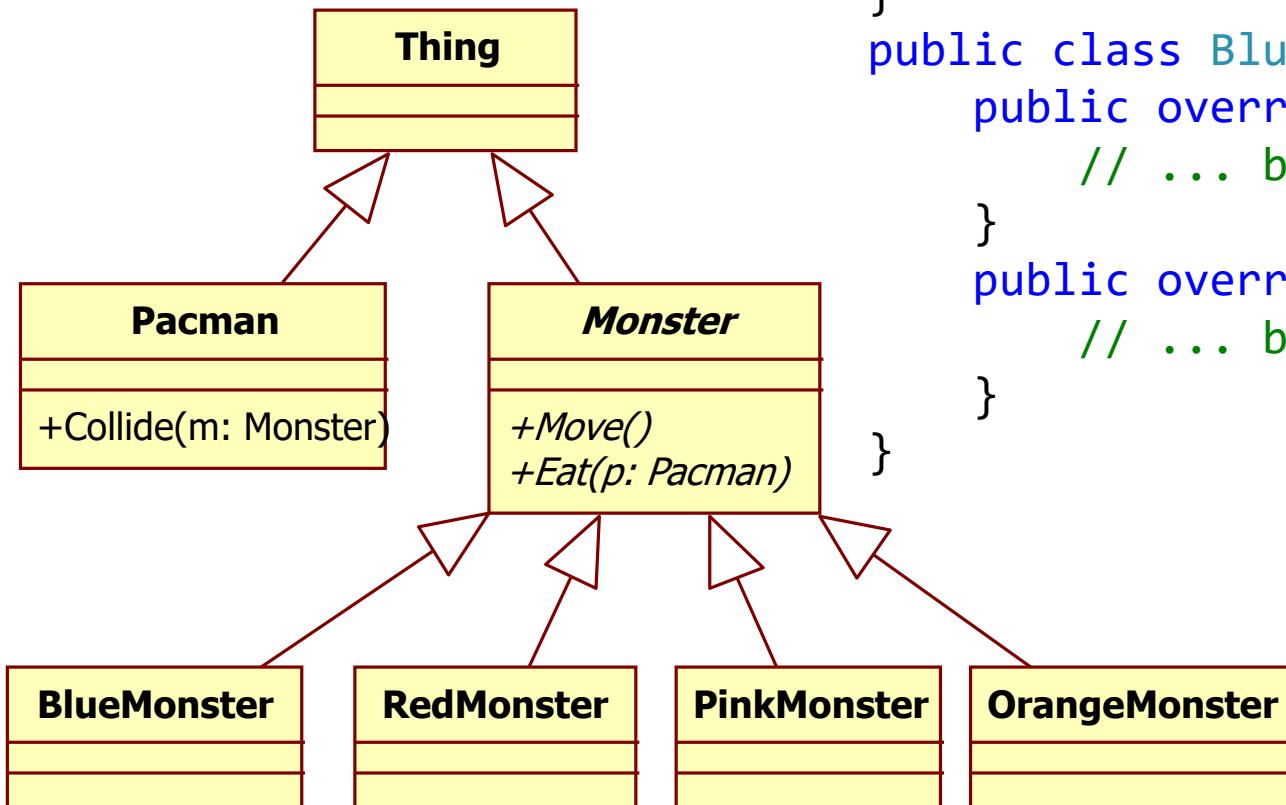
The code shows the implementation of the **Pacman** and **Monster** classes. The **Pacman** class has a `Collide` method that takes a **Monster** object and switches based on its kind. The **Monster** class has a `Move` method that also switches based on its kind.

Problem: dynamic type checking through a type code

I13. Never encode behavior with enum or int values, use polymorphism instead

- Problems if violated:
 - similar to dynamic type test
 - violation of LSP and OCP
- Rules:
 - if methods contain explicit tests for the possible values, then inheritance is necessary
 - don't model objects with common behavior and different states as separate classes
 - use polymorphism only if the behavior is different

I13. Solution



```
public class Pacman : Thing {
    public void Collide(Monster m) {
        m.Eat(this);
    }
}

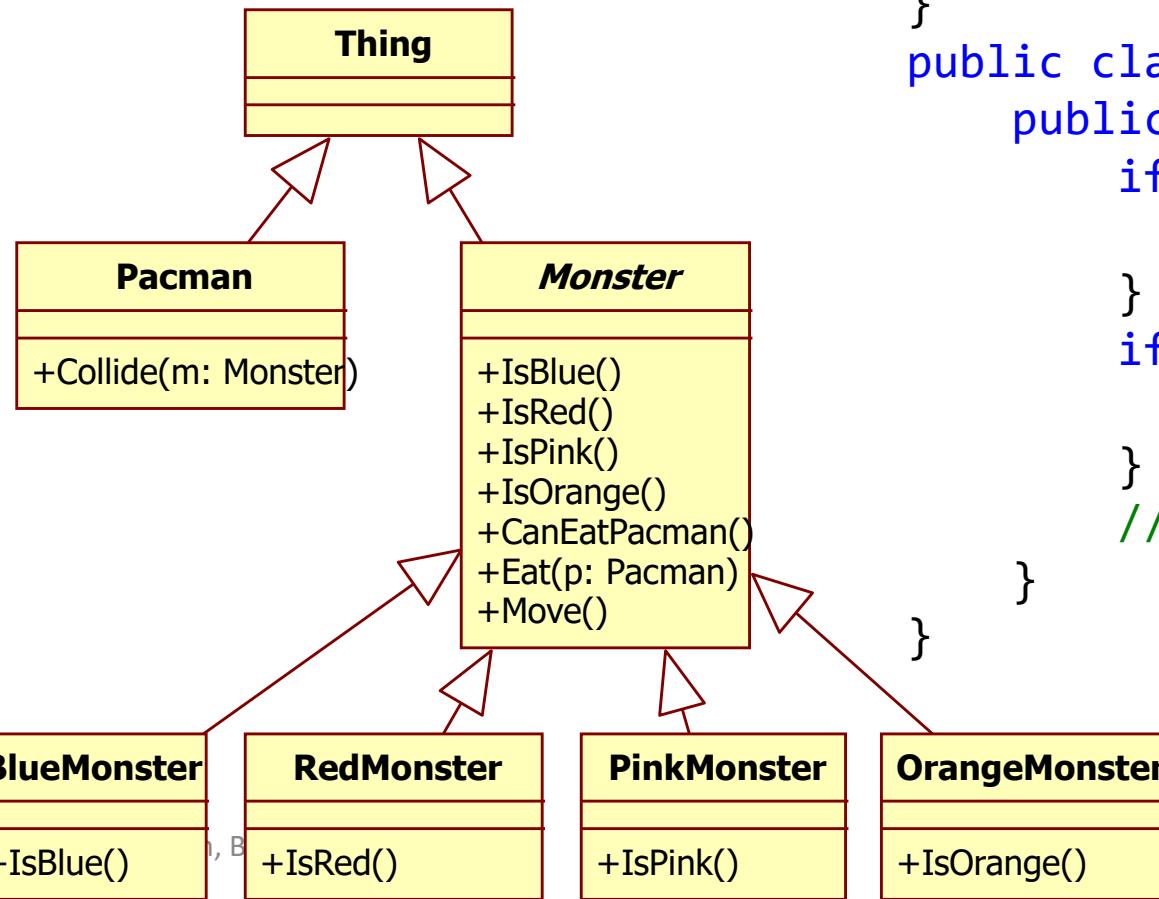
public class BlueMonster : Monster {
    public override void Eat(Pacman p) {
        // ... blue behavior
    }
}

public override void Move() {
    // ... blue behavior
}
```

Different subclasses only if their behavior is different!
Otherwise, just a single Monster class.

I14. Problem

Problem: dynamic type checking
through capability methods



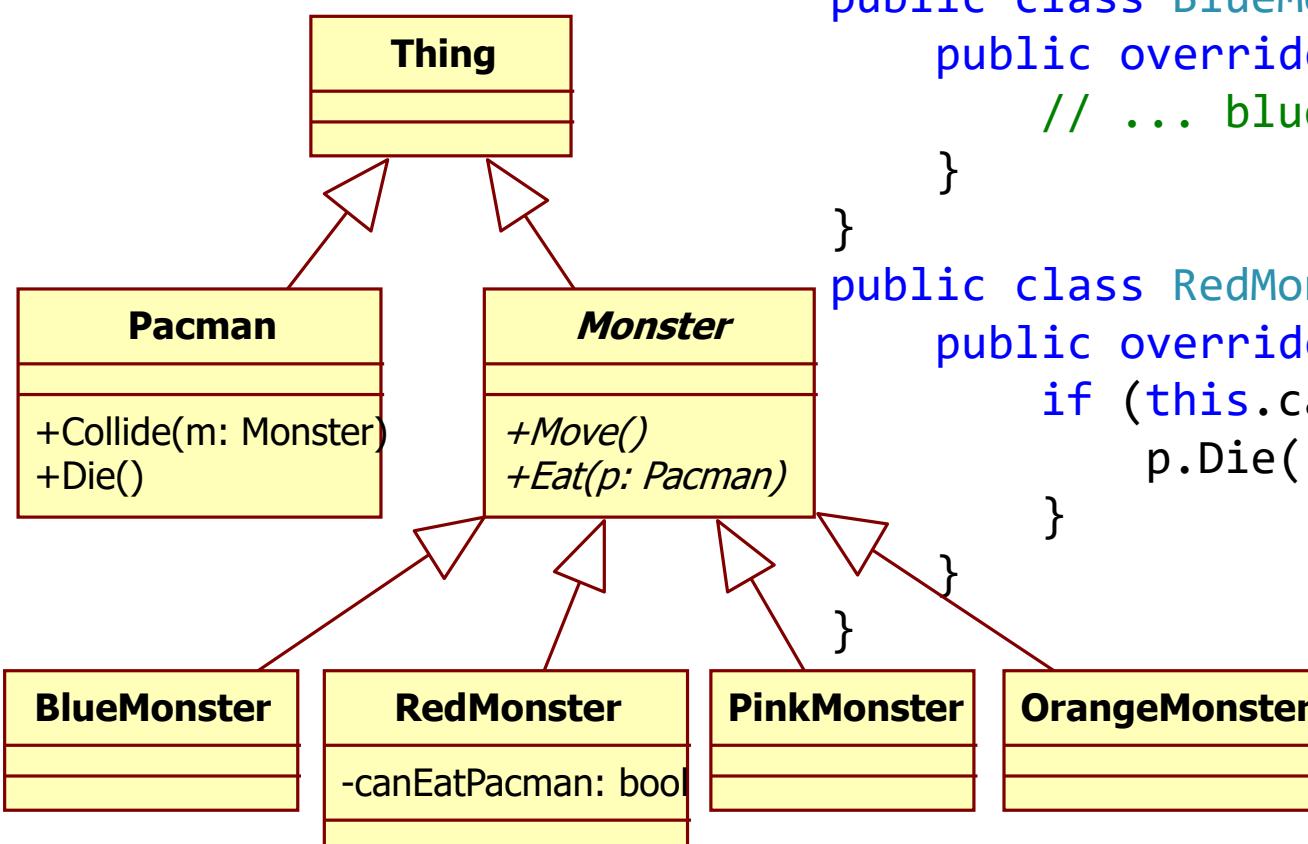
```
public class Pacman : Thing {
    public void Collide(Monster m) {
        if (m.CanEatPacman()) {
            m.Eat(this);
        }
    }
}

public class Monster : Thing {
    public void Move() {
        if (this.IsBlue()) {
            // ... blue behavior
        }
        if (this.IsRed()) {
            // ... red behavior
        }
        // ...
    }
}
```

I14. Don't create type or capability discriminator methods

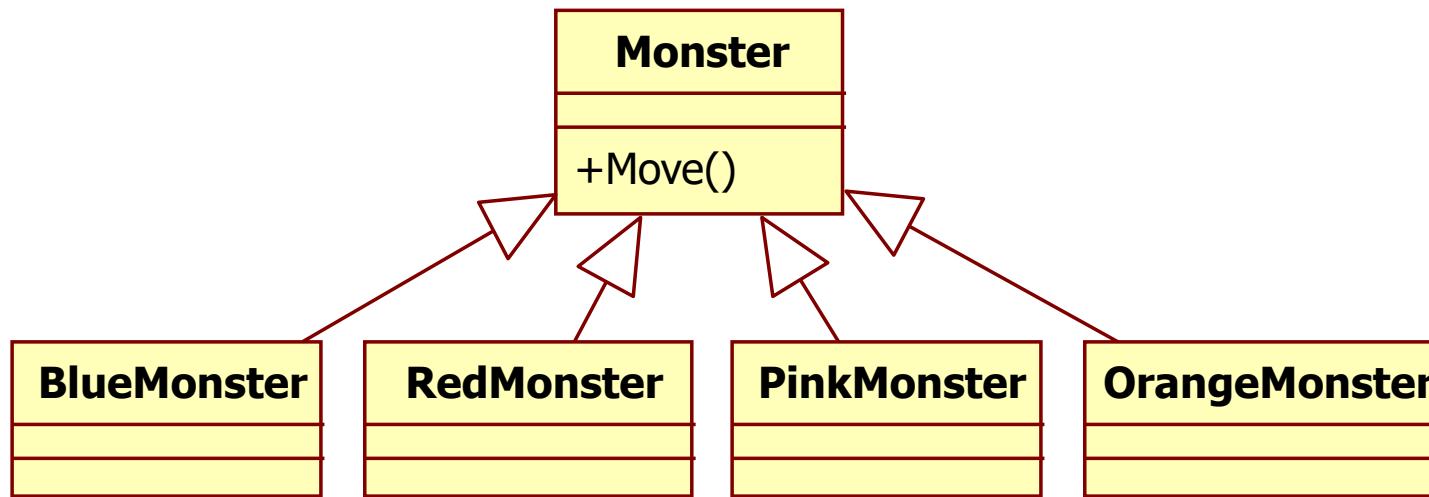
- Problems if violated:
 - violation of TDA
 - responsibility is at the wrong place
- Signs: `is[OfType]()`, `can[DoSomething]()`, etc.
- Rules:
 - use inheritance to achieve different behavior
 - or externalize behavior in a polymorphic solution
 - see Visitor, Strategy, etc. design patterns

I14. Solution



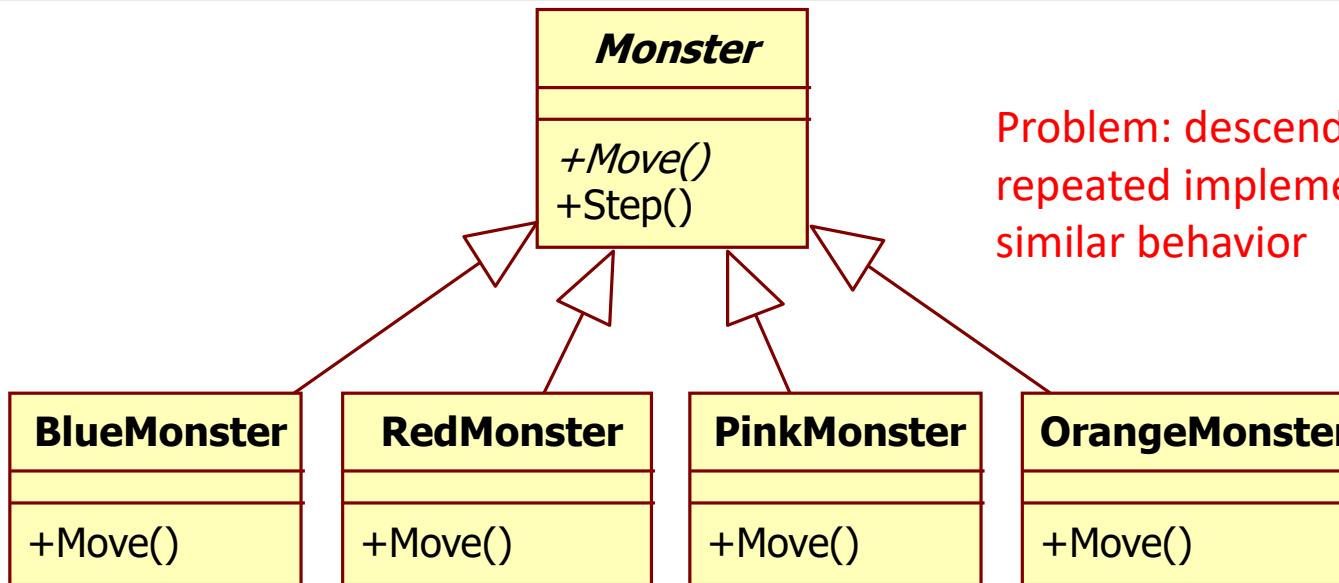
```
public class Pacman : Thing {  
    public void Collide(Monster m) {  
        m.Eat(this);  
    }  
}  
  
public class BlueMonster : Monster {  
    public override void Move() {  
        // ... blue behavior  
    }  
}  
  
public class RedMonster : Monster {  
    public override void Eat(Pacman p) {  
        if (this.canEatPacman) {  
            p.Die();  
        }  
    }  
}
```

I15. Problem I.



Problem: descendants have no added behavior

I15. Problem II.



Problem: descendants have a repeated implementation of a similar behavior

```
public class BlueMonster: Monster {
    public override void Move() {
        this.Step();
    }
}

public class RedMonster: Monster {
    public override void Move() {
        this.Step();
        this.Step();
        this.Step();
    }
}
```

```
public class PinkMonster: Monster {
    public override void Move() {
        this.Step();
        this.Step();
    }
}

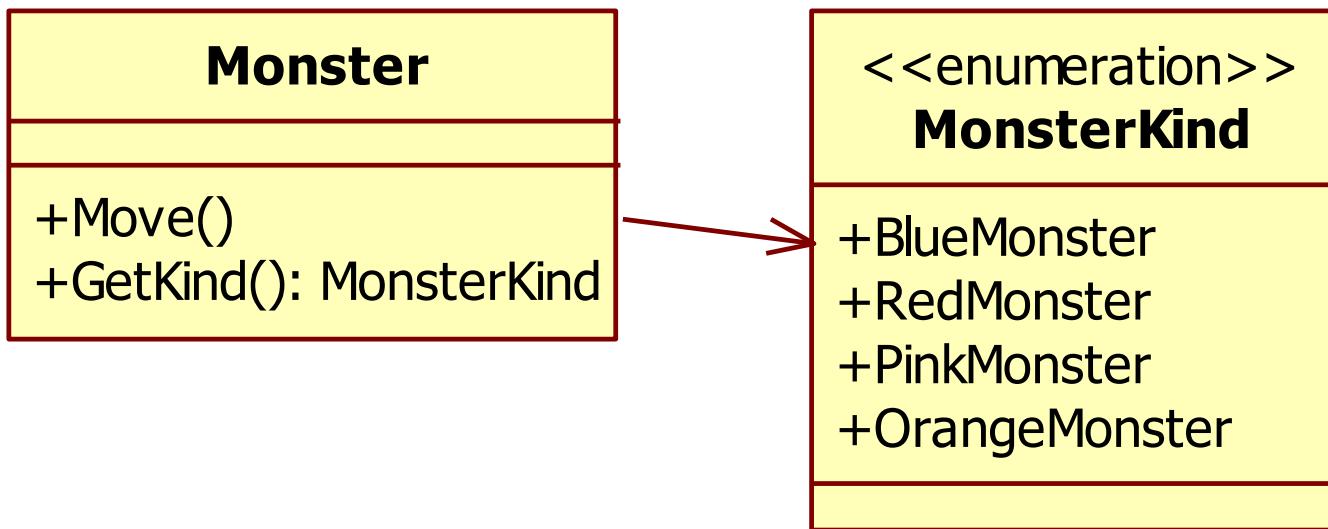
public class OrangeMonster: Monster {
    public override void Move() {
        this.Step();
        this.Step();
    }
}
```

I15. Do not confuse objects with descendants, beware of single instance descendants

- Problems if violated:
 - multiple descendants with the same or similar behavior
 - multiple descendants with no added behavior
- Rules:
 - do not confuse objects with descendants, beware of single instance descendants
 - check whether the descendants really have different behaviors or just different states
 - if the behaviors of the descendants can be united in a single behavior, no descendants are needed

I15. Solution I.

No added behavior: a single class is enough



I15. Solution II.

Similar behavior: a single class is enough,
instances have different states (e.g. speed)

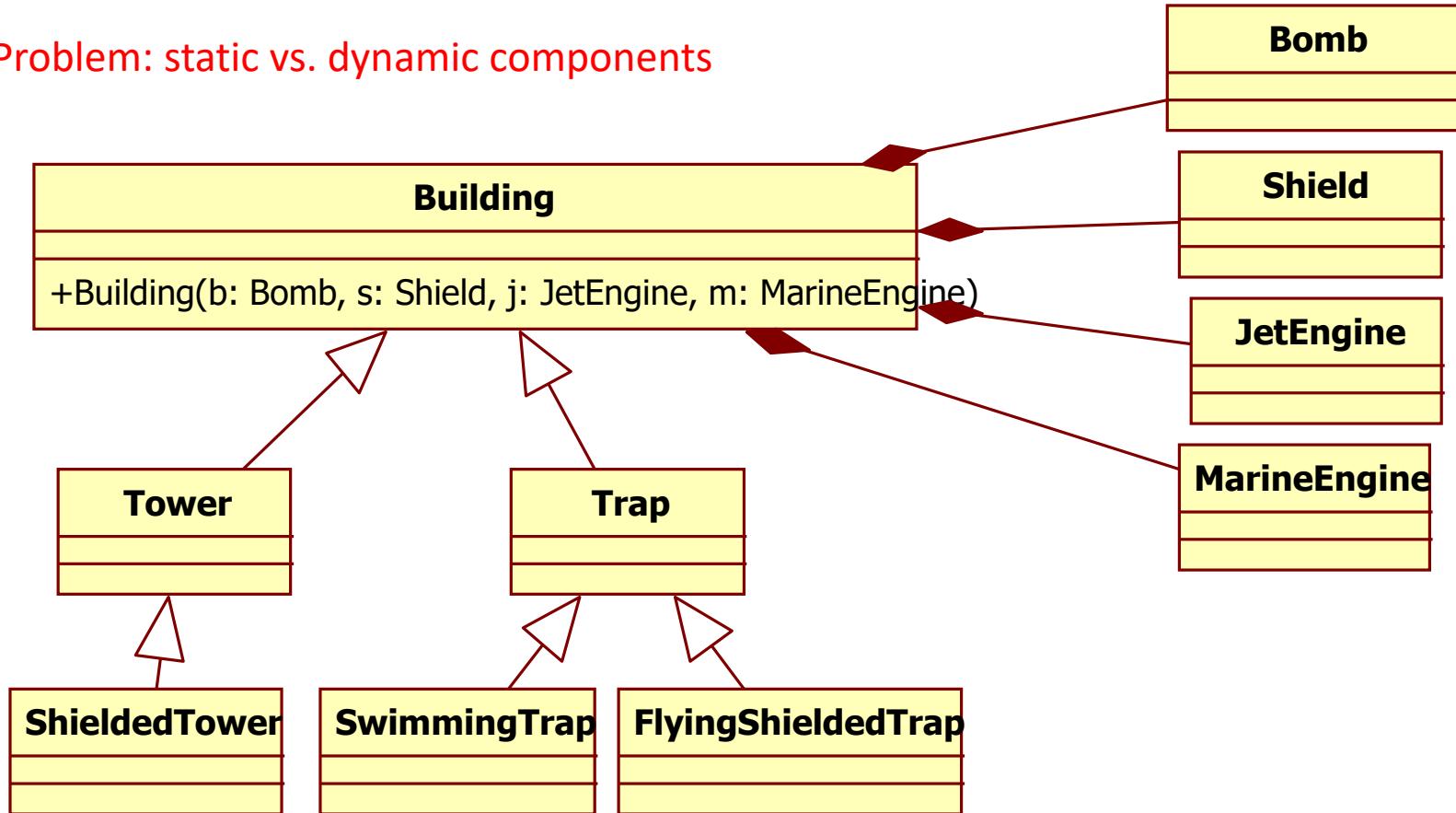
Monster
-speed: int
+Move()
+Step()

```
public class Monster {  
    void Move() {  
        for (int i = 0; i < speed; ++i) {  
            this.Step();  
        }  
    }  
}
```

I16-20. Problem

Problem: what to do with the different combinations of the components?

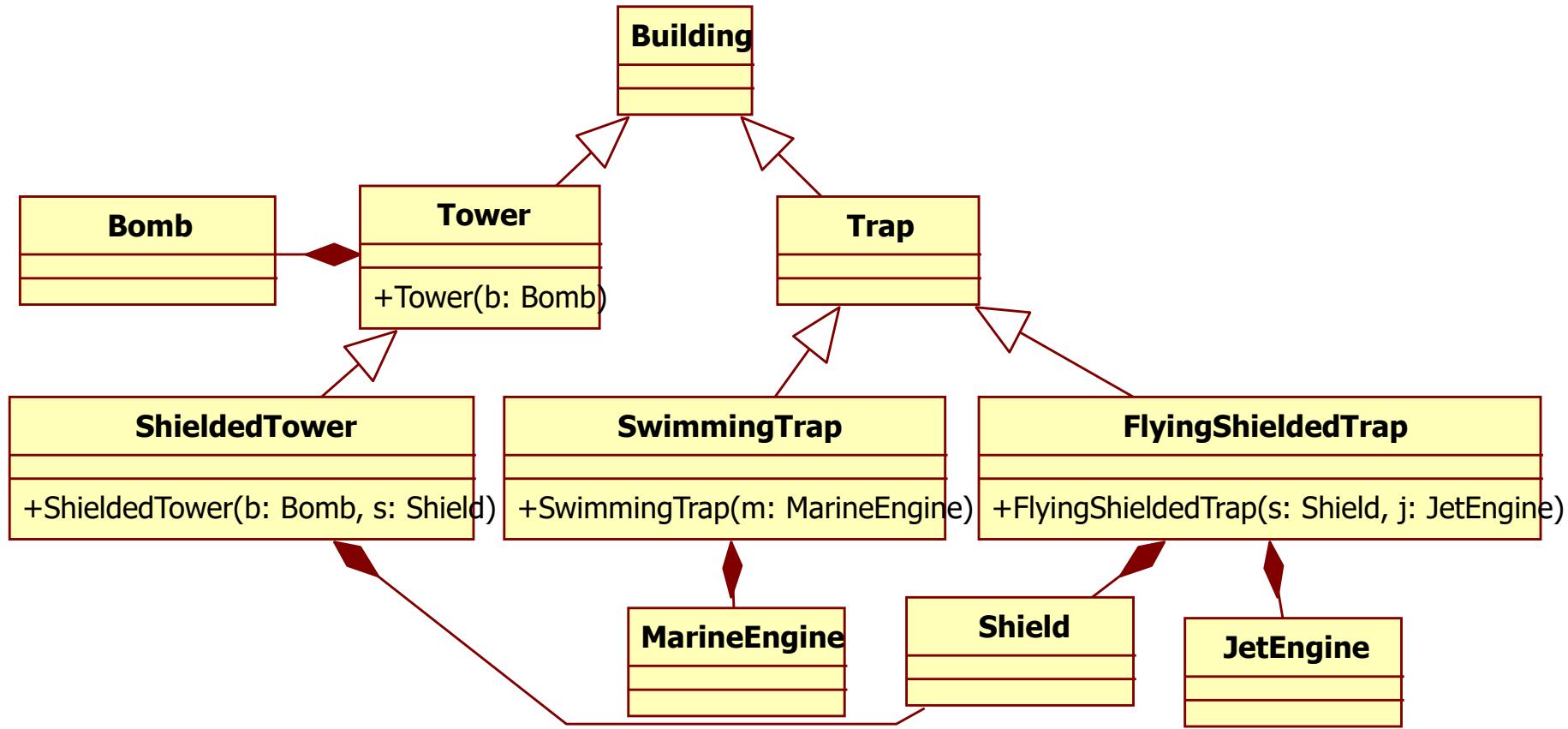
Problem: static vs. dynamic components



I16. Implement static semantics and constraints in the structure of the model

- Problems if violated:
 - it is possible to build models which break the semantics and constraints
- Rules:
 - implement static semantics and constraints in the structure of the model
 - this makes it impossible for the users of the class to break the semantics and constraints
- Exceptions:
 - never do this, if it leads to combinatorial explosion or proliferation of classes
 - in this case, implement semantics and constraints in constructors

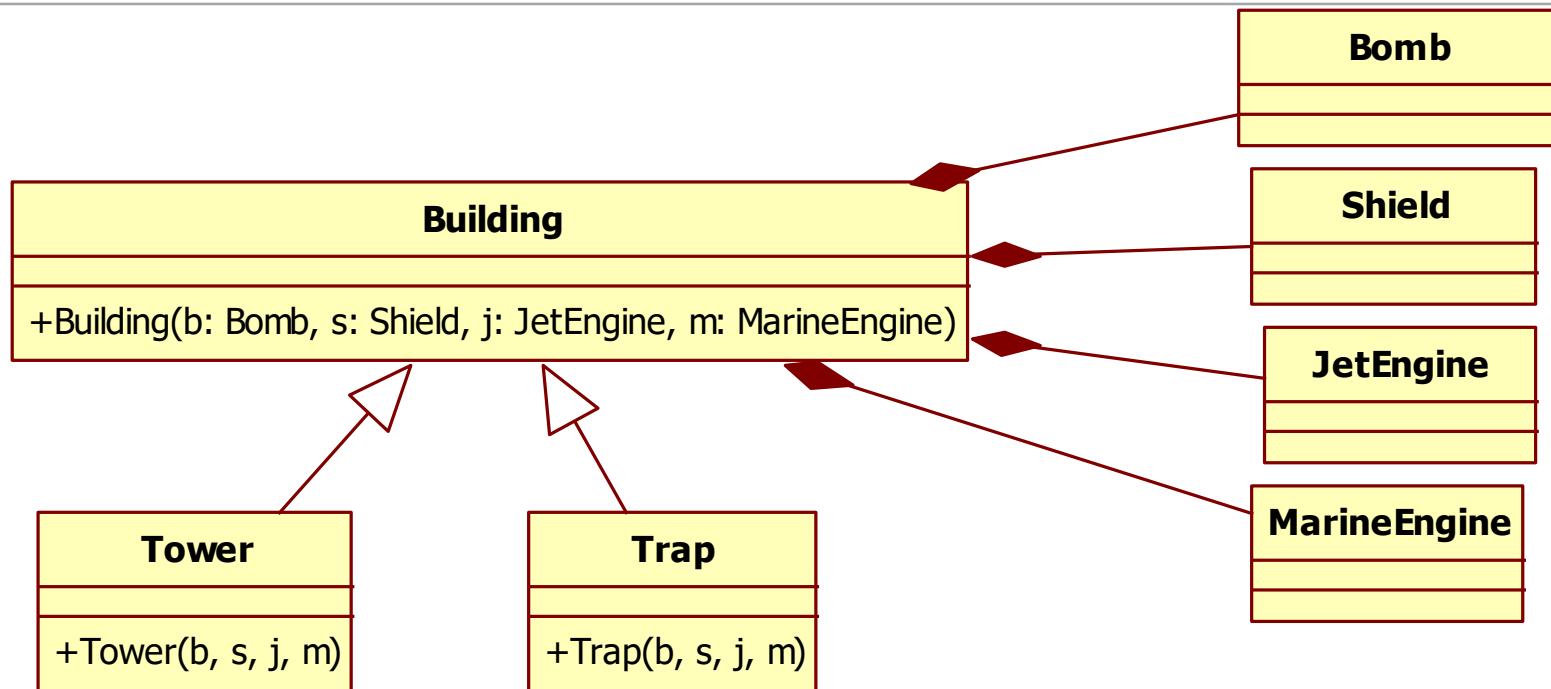
I16. Solution



I17. Implement static semantics and constraints in the constructor

- Problems if violated:
 - it is possible to build models which break the semantics and constraints
- Rules:
 - implement static semantics and constraints in constructors if implementing them in the structure of the model would lead to combinatory explosion
 - put the constraints as deep in the inheritance hierarchy, as possible
 - but do not violate DRY

I17. Solution

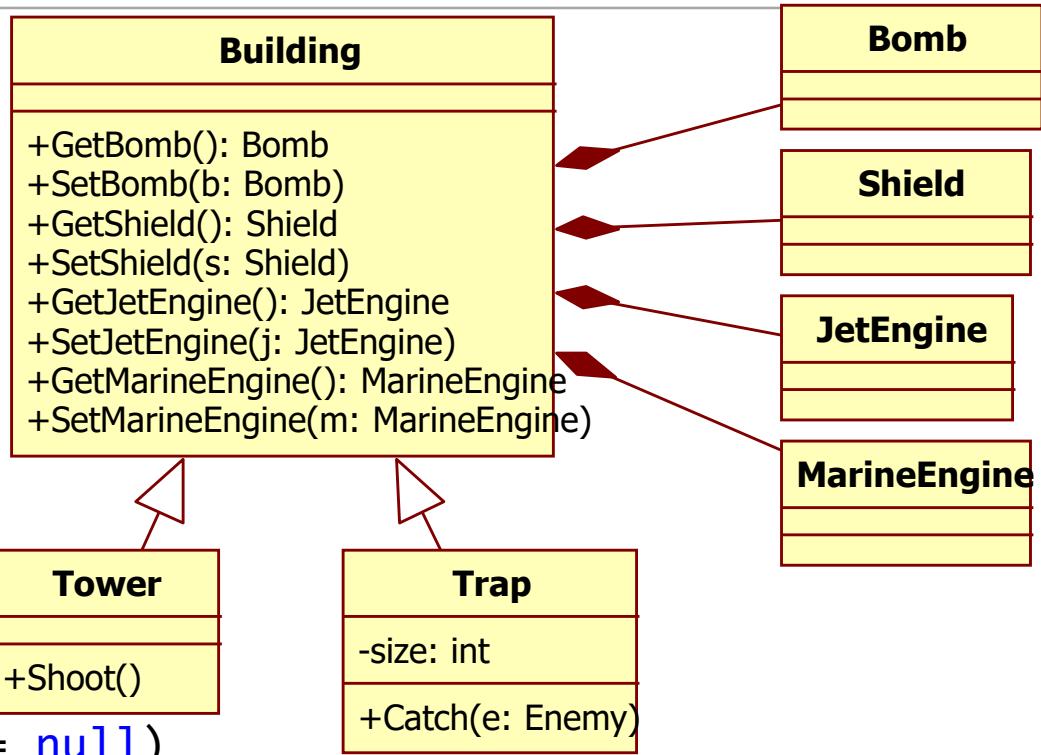


```
public class Trap {
    public Trap(Bomb b, Shield s, JetEngine j, MarineEngine m) {
        if (b != null) throw new Exception("Traps do not shoot bombs.");
    }
}
public class Tower {
    public Tower(Bomb b, Shield s, JetEngine j, MarineEngine m) {
        if (j != null) throw new Exception("Towers cannot fly.");
        if (m != null) throw new Exception("Towers cannot swim.");
    }
}
```

I18. Implement dynamic semantics and constraints in behavior

- Problems if violated:
 - objects can go into inconsistent states
- Rules:
 - implement dynamic semantics and constraints in behavior
 - for example, checking if state is valid at the beginning of methods

I18. Solution

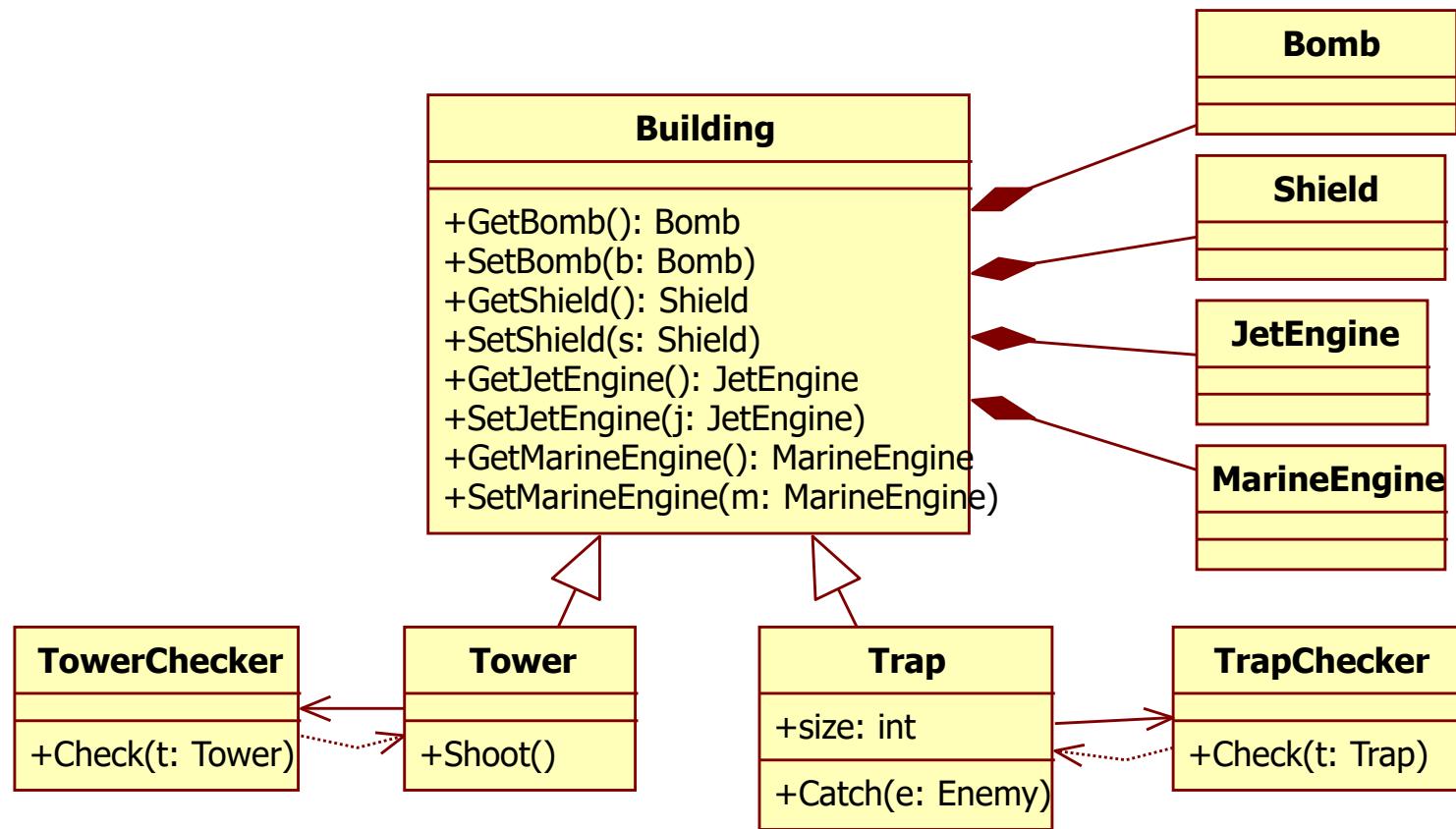


```
public class Tower {  
    public void Shoot() {  
        if (this.GetBomb() == null)  
            throw new Exception("The tower has no bombs.");  
    }  
}  
  
public class Trap {  
    public void Catch(Enemy e) {  
        if (this.size >= 5 && this.GetMarineEngine() != null)  
            throw new Exception("A trap of size 5 cannot swim.");  
    }  
}
```

I19. Implement frequently changing dynamic semantics and constraints in external behavior

- Problems if violated:
 - the class is littered with constraint checking
 - the class must be changed frequently
- Rules:
 - if the constraints are expected to change frequently, then store them in a separate entity
 - for example, in a table or external rules
 - see design patterns: Strategy, Command, Chain of responsibility, Visitor
 - if the constraints are stable, distribute them in the hierarchy (see the rule I18)

I19. Solution



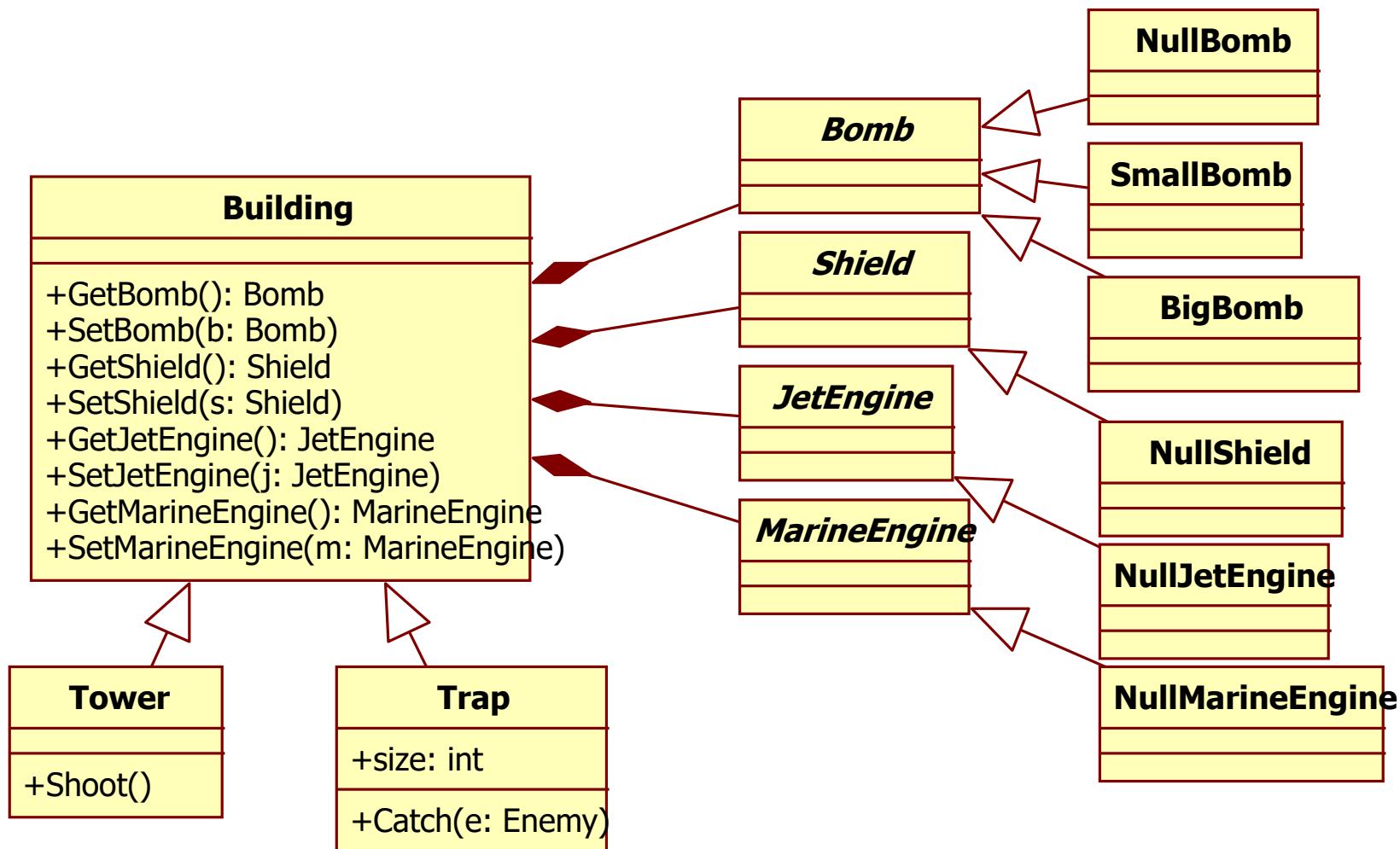
```
public class Tower {  
    public void Shoot() {  
        this.towerChecker.Check(this);  
    }  
}
```

```
public class Trap {  
    public void Catch(Enemy e) {  
        this.trapChecker.Check(this);  
    }  
}
```

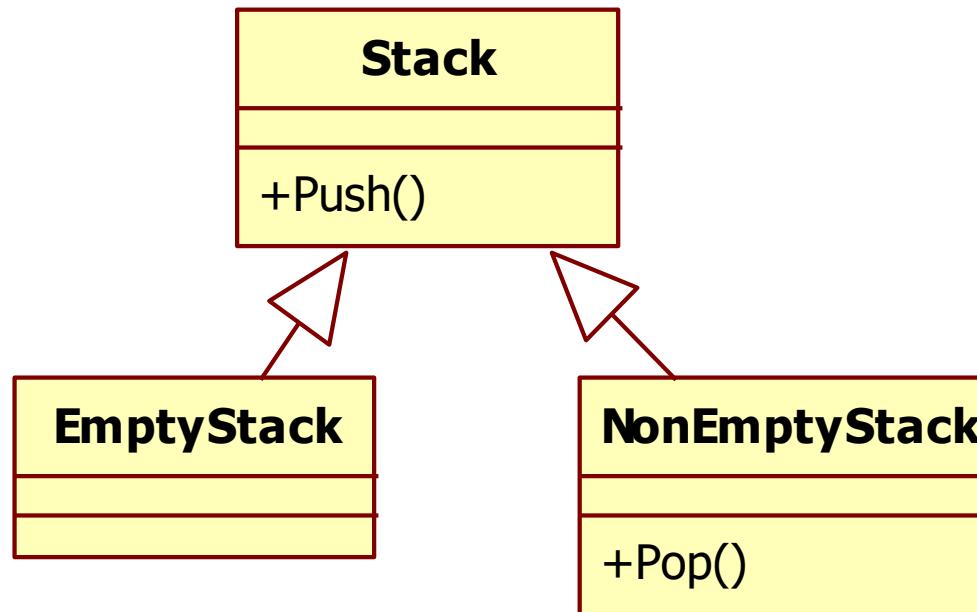
I20. Model optional elements as containment and never as inheritance

- Problems if violated:
 - combinatorial explosion of classes
- Rules:
 - optional elements should be modeled as containment
 - use NullObjects to avoid large if-else constructs resulting from availability checking
 - NullObjects are placeholder objects which do nothing or do a default behavior
 - NullObjects are usually singletons
 - if only specific combinations are allowed, use rules I18, I19

I20. Solution



I21. Problem

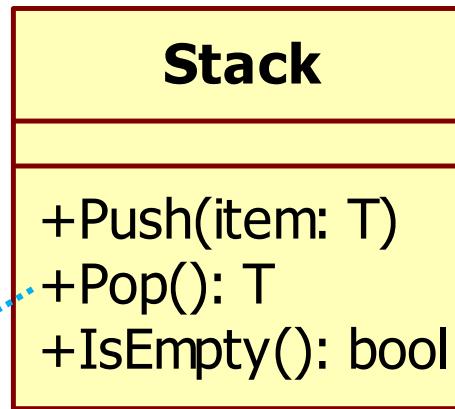


Problem: toggling of types

I21. Do not confuse dynamic constraints with static constraints

- Problems if violated:
 - toggling of types
- Rule:
 - do not confuse dynamic constraints with static constraints, because it leads to toggling of types
 - check dynamic constraints in behavior (I18) or in an external entity (I19)

I21. Solution

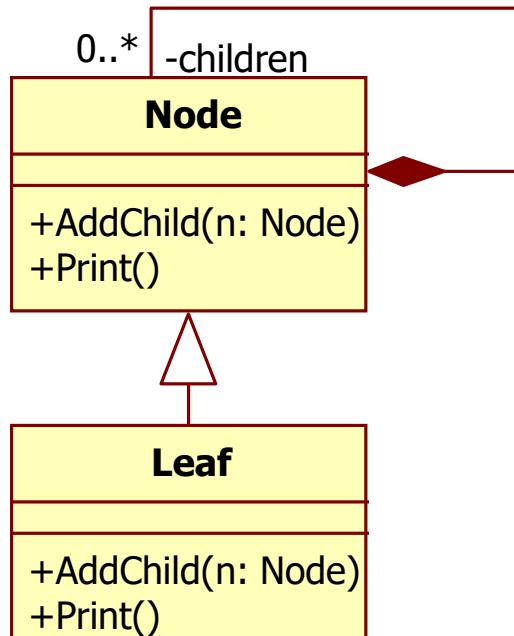


```
T Pop() {  
    if (this.IsEmpty()) throw new EmptyStackException();  
    // ...  
}
```

I22. If you need reflection consider modeling classes instead of objects

- Problems if violated:
 - modeling at the wrong abstraction level
 - performance loss
- Rules:
 - use reflection only if the meta-level is part of the application logic
 - serialization, ORM, configuration, etc.
 - otherwise, the classes created by reflection are in fact objects
 - typical application: report generator
 - consider generalizing these objects into classes

I23. Problem



```
public class Node {  
    public void AddChild(Node n) {  
        this.children.Add(n);  
    }  
}
```

```
public class Leaf {  
    public void AddChild(Node n) {  
        // nop  
    }  
}
```

Problem: overriding with empty behavior

I23. If you need to override a method with an empty implementation the inheritance hierarchy is wrong

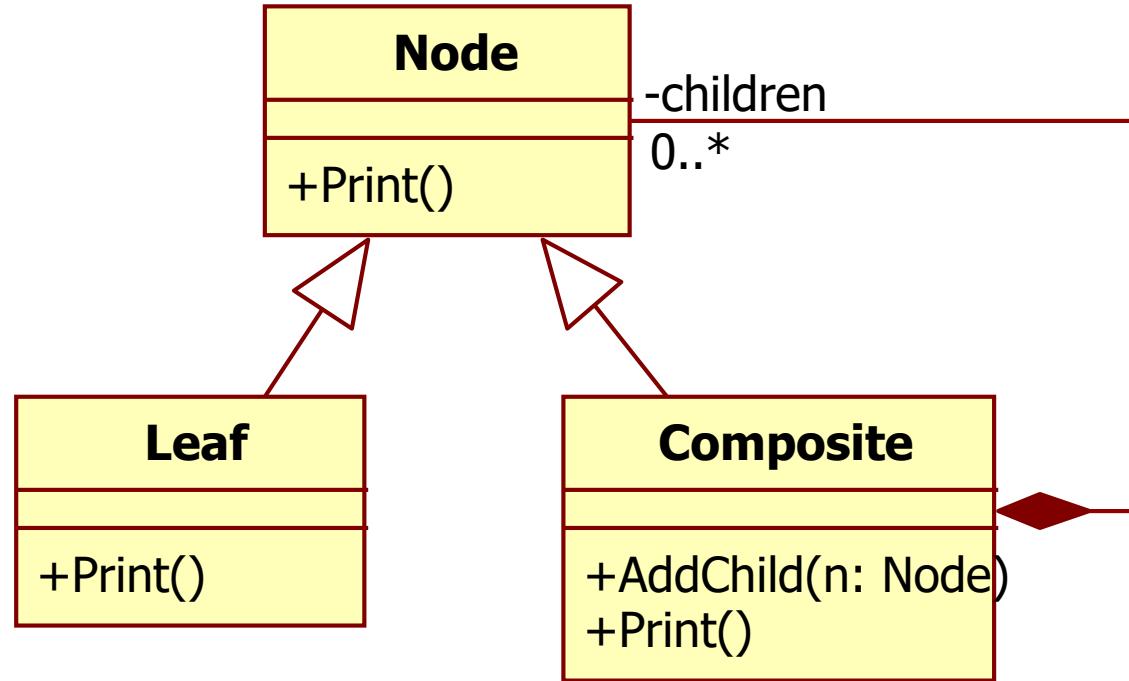
- Problems if violated:

- the base class has too many responsibilities
- the base class implements many combinations of behaviors
- the inheritance hierarchy is mixed up
- violation of LSP

- Rules:

- don't override existing behavior with no behavior
- consider exchanging the base class and the derived class in the inheritance hierarchy with each other if this rule is about to be violated

I23. Solution



I24. Build reusable API instead of reusable classes

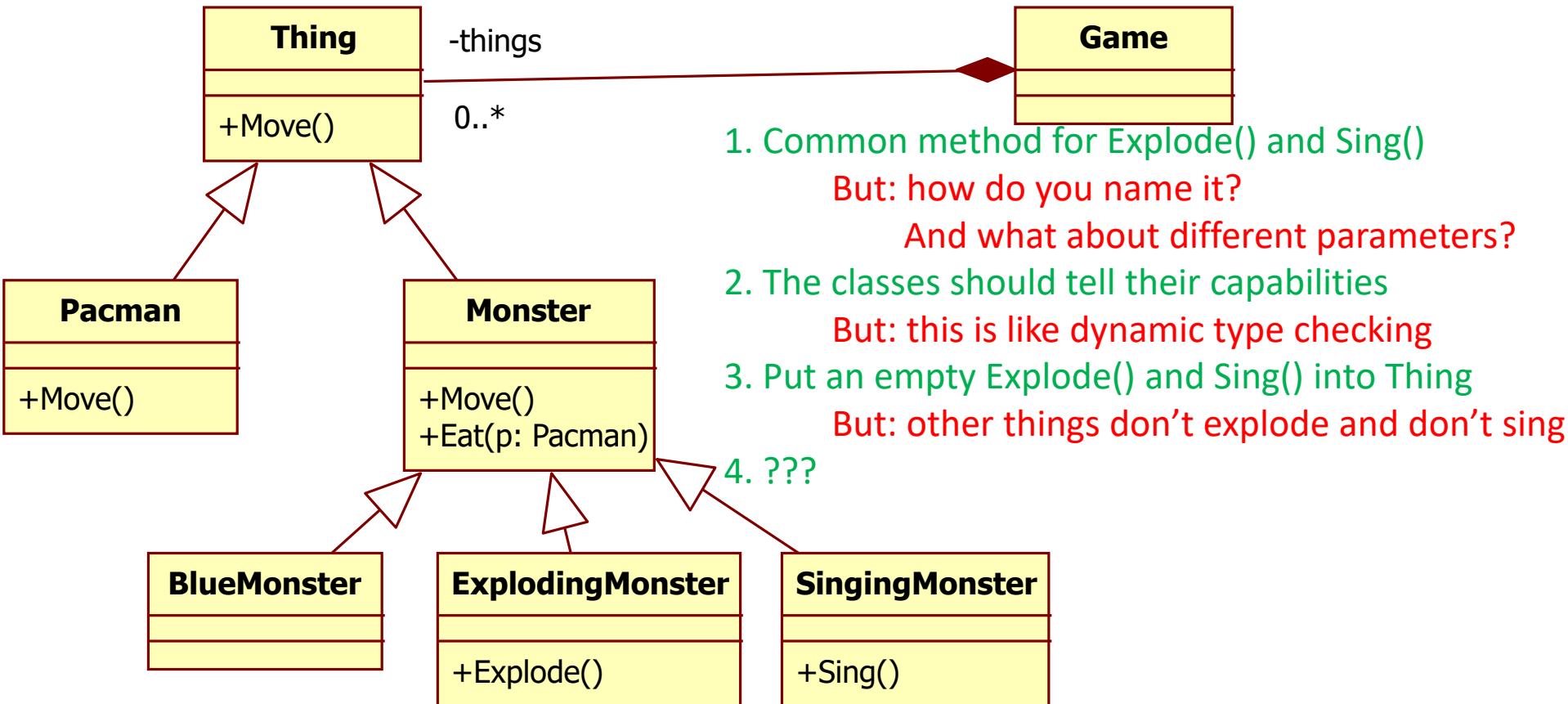
- Problems if violated:
 - modules may not be reused in other projects
 - changes in requirements may result in large changes in the design
- Rules:
 - consider designing a general solution instead of a specific one
 - try to adhere to API design guidelines for more convenient use of modules
 - but keep the time and financial constraints
 - don't overgeneralize

I25. If you need multiple inheritance reconsider your design

- Problems if violated:
 - inheritance hierarchy may be wrong
 - inheritance is used instead of containment
 - duplicate data
- Rules:
 - check the inheritance order of the base classes
 - check whether containment is more appropriate
 - ask the inheritance-containment questions
 - Is A a kind of B? Is A a part of B?
 - if you have multiple inheritance, assume you are wrong and prove otherwise
 - use multiple inheritance only if you have a good reason for it

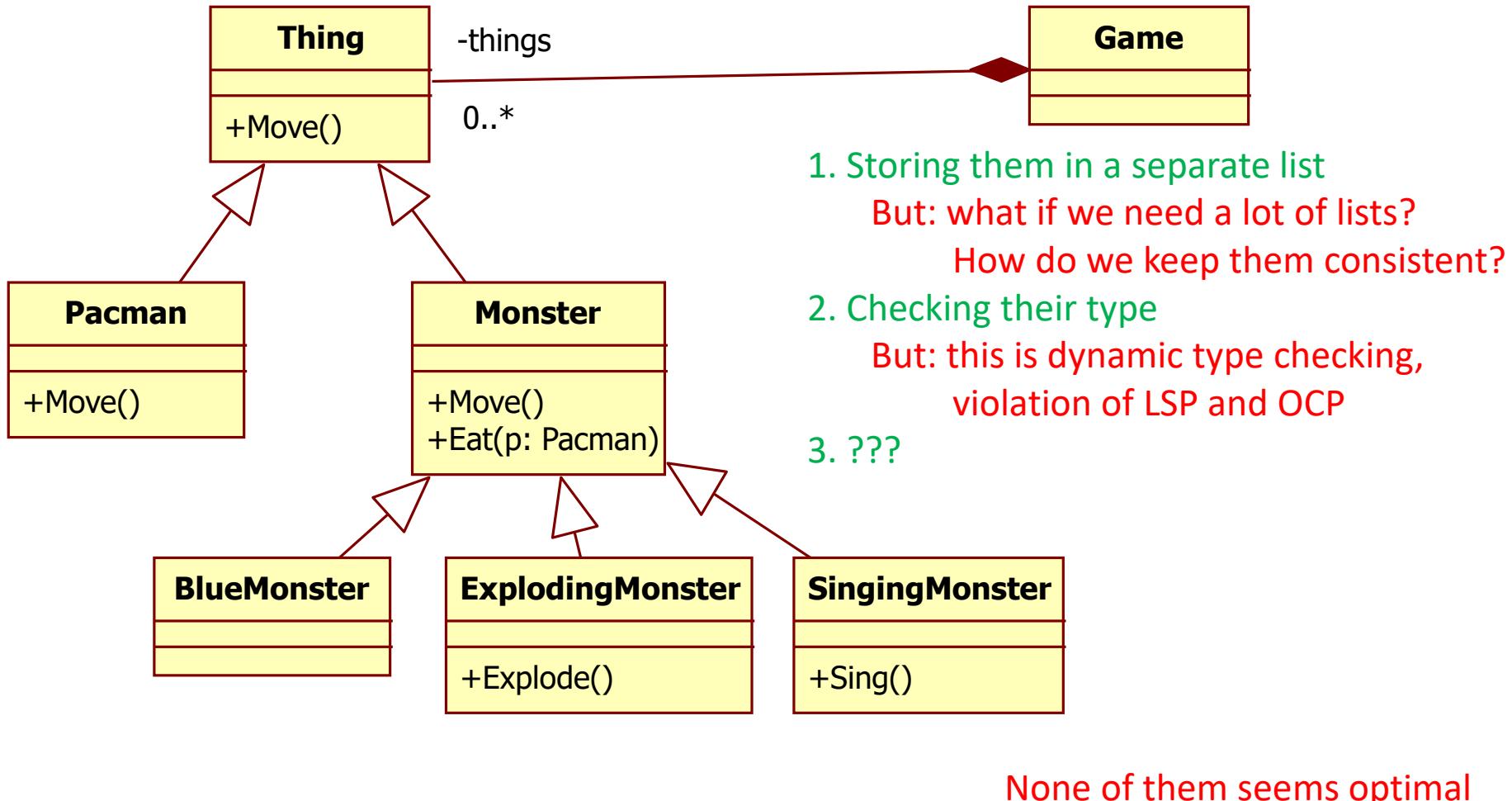
Heterogenous collection problem

How to make the ExplodingMonster explode and the SingingMonster sing?



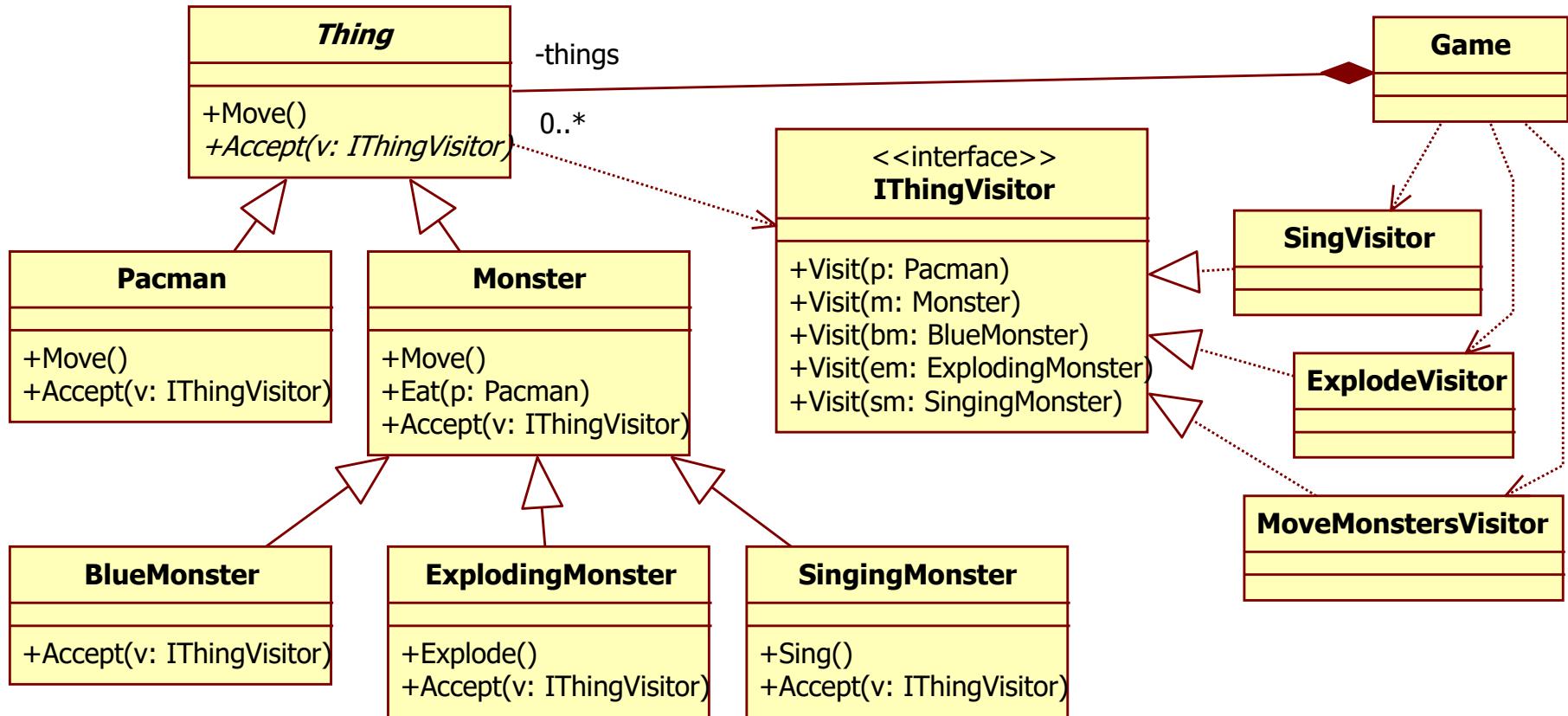
Heterogenous collection problem

How to move only the Monsters?



Heterogenous collection solution

Visitor design pattern



Advantage: although Visitors still violate OCP, this can be recognized at compile time
Possibly an even better solution: Acyclic Visitor pattern, it does not violate OCP at all

Disclaimer

Disclaimer

- OO design principles and heuristics are like pirate code: they are “*more what you’d call “guidelines” than actual rules*” (Barbarossa, Pirates of the Caribbean)
- Some of them even contradict each other
- Don’t fight religious wars over OO purist issues for hours

Disclaimer

- Try to keep the guidelines as much as you can, but if you cannot find a better solution, you can break them
- We are engineers: we have to make tradeoffs
- The important thing is that the software should work in the end
- Disclaimer of the disclaimer: this is not an excuse for not thinking and for ignoring the guidelines

Refactoring

Objektumorientált szoftvertervezés

Object-oriented software design

Dr. Balázs Simon

BME, IIT

Outline

- What is refactoring?
- Code smells
- Refactoring techniques

What is refactoring?

What is refactoring?

Refactoring (noun):

*a change made to the internal structure of software to make it **easier to understand** and **cheaper to modify** without changing its observable **behavior***

Refactoring (verb):

***to restructure** software by applying a series of refactorings without changing its observable behavior*

(Martin Fowler)

When do we refactor?

- We need to add a feature to a program
 - but it is inconvenient: the code is not well structured
- Well structured program:
 - it is convenient to add a new feature
 - easy to maintain
- If the feature is inconvenient to add:
 - 1. Refactor the program, to make it easier to add the feature
 - 2. Add the feature

When to refactor?

- Rule of thumb: Three Strikes and You Refactor
 - *The first time you do something, you just do it.*
 - *The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway.*
 - *The third time you do something similar, you refactor.*
- When adding a function
 - maintains code structure and good design
- When needing to fix a bug
 - clearer structure helps understanding
- When doing code review
 - makes code more clear to review

Refactoring vs. adding a new feature

- Refactoring:
 - is about changing existing code
 - no new functionality
 - only makes the code better
- Adding a feature:
 - existing code is not changed
 - usually just extending the program with new modules, classes or methods

Rules of refactoring

- Must have a solid set of tests
 - functional requirements must be kept
 - refactoring changes code:
 - bugs can happen
 - human error
 - automatic tests are best
 - helps repeating them without effort
- Take small steps
 - bugs are easier to find
 - check often: unit testing
- Have no fear of changing names
 - code must be understood by humans
 - good IDE helps with changes

Steps of refactoring

- 1. Have a solid set of unit tests
 - use existing ones
 - create new ones
- 2. Make sure the tests pass on the old code
- 3. Make a small change
 - easier to test
 - the changed code must be (more) readable
- 4. Run tests on the changed code
 - do not start other changes until the new code passes the tests
- 5. Repeat for other changes from step 1.

Advantages of refactoring

- Improves design
 - design decays: with each modification the code gets worse
 - refactoring helps to keep the structure
- Makes the code easier to understand
 - code is more read than written
 - people will have to maintain it
- Helps finding bugs
 - for the code to be refactored it must be understood
 - during rewriting bugs can emerge
- Helps faster programming
 - sounds counterintuitive
 - without good design no fast change can be made

Problems of refactoring

- Databases
 - tables are rigid
 - code might rely on them
 - object-relational mapping layer might be needed
- Interfaces
 - be careful with changing public interface
 - retain and support the old interface for a while
 - mark the old one deprecated
 - don't publish interfaces prematurely

Code smells

Code smells

- How to find code needing refactoring?
 - No clear criteria
- Code smells are close
 - bad designs that catch attention
 - identification is half victory: solution is usually easy or trivial
- *A code smell is a surface indication that usually corresponds to a deeper problem in the system (Martin Fowler)*
- *Smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality*
- Code smells are not bugs:
 - they are not technically incorrect
 - they do not prevent the program from functioning
- But:
 - they just indicate weaknesses in design
 - they may slow down development
 - they may increase the risk of bugs and failures in the future

S1. Duplicated code

- Description:
 - identical or very similar code exists in more than one location
- Problem:
 - violation of DRY
 - usually a violation of TDA
 - modification is error-prone
- Refactor:
 - Extract method
 - Extract class

S2. Long method

- Description:
 - method is too long
 - too many branches or loops
- Problem:
 - difficult to understand
 - difficult to modify
- Refactor:
 - Split into multiple methods
 - e.g. along comments: they indicate semantic distance between blocks
 - Introduce method object
 - create a class with multiple methods from the long method
 - Decompose conditionals, loops, blocks into methods

S3. Long parameter lists

- Description:
 - method has too many (>3) parameters
- Problem:
 - difficult to pass parameters from the client code
 - difficult to understand
- Refactor:
 - Split into multiple methods
 - Change long parameter lists to parameter objects

S4. Large class

- Description:
 - class has too many methods
- Problem:
 - class has too much responsibility -> violation of SRP
 - probably it is a god-class
 - clients probably don't use all the methods -> violation of ISP
- Refactor:
 - Split up the class into smaller classes
 - Consider creating an inheritance hierarchy
 - Distribute the responsibilities among other classes
 - Use ISP

S5. Divergent change

- Description:
 - class is changed from variation to variation
 - e.g. different UI technologies, different DB drivers, etc.
- Problem:
 - might cause unnecessary changes in other parts of the code
- Refactor:
 - Separate variety-specific part into a class: unstable classes
 - Non-changing parts into different class: stable classes
 - let the rest of the code depend only from the stable classes

S6. Shotgun surgery

- Description:
 - a change results in many small alterations in other classes
 - e.g. changing units from imperial to metric, changing literal values, etc.
 - excessive use of literals
- Problem:
 - changes are error-prone
 - violation of DRY
- Refactor:
 - Put all changes into single class
 - e.g. create a constant for literal values
 - Create new class if necessary
 - e.g. create a utility class
 - Literals should be coded as named constants, to improve readability and to avoid programming errors
 - Literals can and should be externalized into resource files

S7. Feature envy

- Description:
 - a method is too interested in an other class
 - usually interest concerns data
- Problem:
 - responsibility at the wrong place
 - high coupling between the method and the class
- Refactor:
 - Move the method to the other class: higher cohesion
 - Extract the relevant part of the method and put into the other class: lower the coupling
 - Put things together that change together: high cohesion

S8. Data clumps

- Description:
 - data items group naturally
 - e.g. people's name, age, etc.
 - same group of parameters across multiple method calls
- Problem:
 - procedural design, not OO
 - leads to data classes + god classes
- Refactor:
 - Encapsulate data into classes
 - Create parameter object
 - Look for methods in other classes (feature envy)

S9. Primitive obsession

- Description:
 - data is stored in primitive types instead of classes
- Problem:
 - not OO, can not be extended easily
 - behavior cannot be attached to primitive types
- Refactor:
 - Replace groups of primitive data with class(es)
 - Replace type code with class (inheritance)
 - Replace type code with state/strategy
 - Replace array of different items with object

S10. Switch statements

- Description:
 - switch statement in code
 - null-checking
- Problem:
 - usually leads to code duplication
 - misplaced responsibility
- Refactor:
 - Replace type code with class (inheritance)
 - Replace type code with state/strategy
 - Introduce NullObject

S11. Parallel inheritance hierarchies

- Description:
 - every time you create a new subclass, you also have to make a subclass of another
 - usually the classes of an inheritance hierarchy have the same prefixes/suffixes
 - subcase of shotgun surgery
- Problem:
 - leads to dependent modifications, duplications
- Refactor:
 - Move methods and fields from the referring hierarchy to the other: the referring hierarchy disappears
- Caution:
 - parallel hierarchies may be a deliberate design decision (e.g. simulating multiple inheritance)

S12. Lazy class

- Description:
 - class is not doing enough
 - could have been downsized by refactoring
 - could have been added because of changes that were planned but not made
- Problem:
 - overkill to maintain
- Refactor:
 - Eliminate it
 - Inline the class
 - If it is a subclass, collapse the hierarchy

S13. Speculative generality

- Description:
 - "we might need this ability someday"
 - heavy extension machinery which is not used
 - code only used by tests
 - forced usage of overcomplicated design patterns where simpler design would suffice
- Problem:
 - too much unnecessary code to maintain
 - violation of YAGNI
- Refactor:
 - Get rid of the unused heavy machinery
 - Collapse hierarchy
 - Unused parameters should be removed

S14. Temporary field

- Description:
 - an attribute is set only in certain circumstances
 - e.g. object scope "global" helper variables
 - e.g. variables used only in some of the methods running a complex algorithm, but not in others
- Problem:
 - difficult to understand and maintain
 - an object does not use all of its variables
 - low cohesion
- Refactor:
 - Extract such attributes to new class
 - with relevant methods as well
 - Introduce NullObject to eliminate conditional code

S15. Message chains

- Description:
 - too long method chains
 - e.g. getA().getB().getC()....
- Problem:
 - client is coupled to the structure of the navigation
 - change to the intermediate relationships causes the client to have to change
 - violation of LoD
- Refactor:
 - Hide delegation
 - Move methods between classes

S16. Middle man

- Description:
 - too much delegation a class to another
- Problem:
 - delegation overhead
- Refactor:
 - Remove the middle man (talk to the target class directly)
 - Inline methods
 - Replace delegation with inheritance

S17. Inappropriate intimacy

- Description:
 - classes accessing each other's private members directly
- Problem:
 - responsibilities at the wrong place
 - too much coupling between the classes
- Refactor:
 - Move methods and fields between the classes to reduce coupling
 - Change bidirectional association to unidirectional
 - Let another class act as go-between
 - Replace delegation with inheritance

S18. Alternative classes with different interfaces

- Description:
 - classes for the same task having different interfaces
- Problem:
 - classes are not interchangeable
- Refactor:
 - Rename methods
 - Move methods into other classes if necessary
 - Extract superclass if possible
 - Goal: reach a common interface

S19. Incomplete library class

- Description:
 - library class (server) can not be modified
- Problem:
 - usual refactoring does not work on it, we need to make an adaptor
- Refactor:
 - Introduce new method into client, server is parameter
 - Create new subclass of server with new functionality

S20. Data class

- Description:
 - class with only setter and getter methods
- Problem:
 - not OO, encapsulation is violated
 - class has no responsibility
- Refactor:
 - Remove setting method on read only attributes
 - Move behavior into the data class from clients
 - both whole and partial methods might work
 - Goal: the class has to gain real responsibility

S21. Refused bequest

- Description:
 - subclass doesn't need the superclass functionality
- Problem:
 - not strong smell, but can cause confusion
 - possibly the inheritance order is wrong
 - superclass has unnecessary responsibility
- Refactor:
 - Reorder the inheritance hierarchy
 - Push down method or field into relevant subclass
 - If parent interface is refused, replace inheritance with delegation

S22. Comments

- Description:
 - too much explanation comment in the code
- Problem
 - the code is overcomplicated
- Solution
 - Extract methods and simplify the code
 - Rename method if necessary
 - If comment is needed to clarify what the code is doing, try to refactor
 - comments should say why you did something

S23. Downcasting

- Description:
 - use of type cast
 - use of instanceof
- Problem:
 - a type cast breaks the abstraction model
 - violation of OCP, LSP
- Refactor:
 - The abstraction may have to be refactored or eliminated
 - Move the behavior into the class to which you cast

Refactoring techniques

Composing methods

- F1. Extract method
 - take a piece of code and turn it into a method
- F2. Inline method
 - take a method call and replace it with the body of the method
- F3. Inline temporary
 - if the temporary variable is used only once, get rid of it
- F4. Replace temporary with query
 - extract temporary variable as a method
- F5. Introduce explaining variable
 - replace a complex expression with a temporary variable
- F6. Split temporary variable
 - use separate temporary variables for unrelated assignments
- F7. Remove assignments to parameters
 - use a temporary variable instead of assigning to a parameter
- F8. Replace method with method object
 - extract method with local variables
- F9. Substitute algorithm
 - replace algorithm with a clearer one

Moving features between objects

- F10. Move method
 - move responsibility from one class to another
- F11. Move field
 - move field from one class to another
- F12. Extract class
 - select some fields and methods and create a new class for them
- F13. Inline class
 - eliminate a class by moving its fields and methods into another class
- F14. Hide delegate
 - create a method to prevent call chaining
- F15. Remove middle man
 - get the client to call the delegate directly
- F16. Introduce foreign method
 - put a new method in the client with the server as parameter
- F17. Introduce local extension
 - create a new subclass of the server with the new methods

Organizing data I.

- F18. Self encapsulate field
 - create getter/setter methods for the field
- F19. Replace data value with object
 - turn data item into an object
- F20. Change value to reference
 - turn many equal instances to references (e.g. flyweight)
- F21. Change reference to value
 - turn immutable reference objects to separate instances
- F22. Replace array with object
 - replace the array with an object that has a field for each element
- F23. Duplicate observed data
 - e.g. database – model – GUI layers
- F24. Change unidirectional association to bidirectional
 - two-way administration
- F25. Change bidirectional association to unidirectional
 - drop the unneeded end

Organizing data II.

- F26. Replace magic numbers with symbolic constant
 - name literals as constants
- F27. Encapsulate field
 - make a non-private attribute private and provide accessors
- F28. Encapsulate collection
 - provide add/remove methods, provide read-only view
- F29. Replace record with data class
 - interface with a traditional programming environment
- F30. Replace type code with class
 - multiple classes instead of a type code
- F31. Replace type code with subclasses
 - inheritance and polymorphism instead of a type code
- F32. Replace type code with state/strategy
 - if inheritance cannot be used replace type code with a state/strategy object
- F33. Replace subclass with fields
 - subclasses have no added behavior, move methods that return constant data to the superclass as fields

Simplifying conditional expressions

- F34. Decompose conditional
 - extract methods from the if, then, else parts
- F35. Consolidate conditional expression
 - combine sequence of conditional tests with the same result into a single conditional expression
- F36. Consolidate duplicate conditional fragments
 - move same fragments of code in all branches of an if outside
- F37. Remove control flag
 - use a break or return instead of a control flag variable
- F38. Replace nested conditional with guard clauses
 - replace nested conditionals with a series of if-else constructs
- F39. Replace conditional with polymorphism
 - replace conditional depending on the type of an object with polymorphism
- F40. Introduce null object
 - replace the null value with a null object to avoid null-checks
- F41. Introduce assertion
 - make assumptions explicit with assertions

Making method calls simpler I.

- F42. Rename method
 - change name of the method to reveal its purpose
- F43. Add parameter
 - add a parameter to pass more information from caller
- F44. Remove parameter
 - remove parameter if it is not used anymore
- F45. Separate query from modifier
 - create two methods, one for the querying and one for the modification
- F46. Parameterize method
 - combine similar methods into one method with additional parameters
- F47. Replace parameter with explicit methods
 - split a method with multiple cases to multiple methods with fewer parameters
- F48. Preserve whole object
 - instead of passing parts of an object as multiple parameters, pass the whole object

Making method calls simpler II.

- F49. Replace parameter with method
 - instead of passing the result of one method to another, let the second method call the first method
- F50. Introduce parameter object
 - replace a group of parameters that naturally go together with an object
- F51. Remove setting method
 - set attributes in the constructor, do not provide setter methods
- F52. Hide method
 - if other classes do not use a method, make it private
- F53. Replace constructor with factory method
 - if more than a simple construction is needed, use a factory method
- F54. Encapsulate downcast
 - if the result of a method needs to be downcasted by the clients, move the downcast into the method
- F55. Replace error code with exception
 - throw an exception instead of returning special values
- F56. Replace exception with test
 - instead of catching exceptions check the parameters before passing them to the server

Dealing with generalization I.

- F57. Pull up field
 - move a common field in two subclasses to a superclass
- F58. Pull up method
 - move a common method in two subclasses to a superclass
- F59. Pull up constructor body
 - move the identical parts of the constructors of two subclasses to a superclass
- F60. Push down method
 - if a method is relevant only for a subset of the subclasses, move it to those subclasses
- F61. Push down field
 - if a field is relevant only for a subset of the subclasses, move it to those subclasses
- F62. Extract subclass
 - if a subset of features is relevant only for a subset of the subclasses, create a subclass for those features

Dealing with generalization II.

- F63. Extract superclass
 - if two subclasses have similar features, create a superclass from these features
- F64. Extract interface
 - if several classes have a common interface, extract this subset into an interface
- F65. Collapse hierarchy
 - if the subclass add little or no additional behavior, merge it with its superclass
- F66. Form template method
 - if two subclasses have similar methods performing similar steps in the same order, make a template method from them in the superclass
- F67. Replace inheritance with delegation
 - if the subclass uses only a subset of the superclass's interface, use delegation instead of inheritance
- F68. Replace delegation with inheritance
 - if the class does mostly delegation, make the delegating class a subclass of the delegate

High level refactoring

- F69. Tease apart inheritance
 - if an inheritance hierarchy is doing two jobs at once, split them into two hierarchies
- F70. Convert procedural design to objects
 - move behavior to data classes
- F71. Separate domain from presentation
 - move domain logic from the GUI to separate domain classes
- F72. Extract hierarchy
 - if a class does too much work, create a hierarchy of classes in which each subclass represents a special case

Clean Object-Oriented Design

Objektumorientált szoftvertervezés

Object-oriented software design

Dr. Balázs Simon

BME, IIT

Outline

- What is clean code?
- Guidelines:
 - Selecting meaningful names
 - Writing functions
 - Writing and omitting comments
 - Defining and handling exceptions
 - Objects vs. data structures

What is clean code?

Bad code

- Code will be here for a long time
 - program code in a general purpose language
 - higher abstraction level: domain-specific languages
 - the abstraction level will continue to increase
- Bad code:
 - hard to read
 - hard to understand
 - hard to maintain
 - usual causes:
 - hurry
 - tired of working on the software and wanting it to be over
 - laziness
 - inexperience
 - it will be cleaned up later (=never)

Problems with bad code

- Bad code decreases productivity
 - management adds more staff to increase or maintain productivity
 - but it will just get worse: new people don't understand the code, don't know the original design intentions and have to work under pressure
- Rewriting the whole system takes time
 - the old and new systems must be maintained and developed in parallel under changing requirements
 - by the time the new system is ready, its original developers are gone and it can be a mess again

Who is responsible for bad code?

- Not responsible:
 - tight schedule
 - changing requirements
 - management
- Responsible:
 - we, developers
- Bad code:
 - slows us down
 - makes changing the code harder
 - is not an excuse
- The only way to go fast in the short run and in the long run is to keep the code as clean as possible *at all times*

Clean code

- Code is mostly read, not written
 - time ratio is 10:1 or more
- Clean code is an art:
 - hard to define exactly
 - even if you recognize it, it does not mean you can write it
 - it must be practiced
- Properties of clean code:
 - easy to read: should read like a well written prose
 - easy to understand
 - elegant and efficient
 - minimal dependencies
 - complete error handling
 - focused: does one thing well
 - easy for other people to enhance it

Meaningful names

Problem

Why is the 0th item special?

```
public List<int[]> GetThem() {  
    List<int[]> list1 = new List<int[]>();  
    for (int[] x in theList)  
        if (x[0] == 4)  
            list1.add(x);  
    return list1;  
}
```

What does the function do?

What is in the list?

What does 4 mean?

What to do with the result?

The answers cannot be explicitly seen, but they could be!

Problem

Why write a comment? Why not give a better name to the variable?

```
int d; // elapsed time in days
```

Use intention-revealing names

- Choosing good names takes time but saves more than it takes
- Take care with your names
 - should tell you why something exists, what it does, and how it is used
- Change them when you find better ones
- Everyone who reads your code (including you) will be happier if you do
- If a name requires a comment, then the name does not reveal its intent
- A long descriptive name is better than a short enigmatic name
- IDE helps with long names, no need for abbreviations

Solution

```
public List<int[]> getThem() {  
    List<int[]> list1 = new List<int[]>();  
    for (int[] x in theList)  
        if (x[0] == 4)  
            list1.add(x);  
    return list1;  
}
```



```
public List<int[]> getFlaggedCells() {  
    List<int[]> flaggedCells = new List<int[]>();  
    for (int[] cell in gameBoard)  
        if (cell[Consts.StatusValue] == Consts.Flagged)  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```

Or a more object-oriented solution

```
public List<int[]> getFlaggedCells() {  
    List<int[]> flaggedCells = new List<int[]>();  
    for (int[] cell in gameBoard)  
        if (cell[Consts.StatusValue] == Consts.Flagged)  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```



```
public List<Cell> getFlaggedCells() {  
    List<Cell> flaggedCells = new List<Cell>();  
    for (Cell cell in gameBoard)  
        if (cell.isFlagged())  
            flaggedCells.add(cell);  
    return flaggedCells;  
}
```

Solution

```
int d; // elapsed time in days
```



```
int elapsedTimeInDays;
```

← No comment is necessary!

Other examples:

```
int daysSinceCreation;  
int daysSinceModification;  
int fileAgeInDays;
```

Notice the units!

We don't need to read the documentation or to write additional comments.
The code documents itself.

Problem

```
string xml; // example ← XML means something else
```

```
Set<Account> accountList; ← this is not a list
```

```
class XYZControllerForEfficientHandlingOfStrings  
{  
}
```

hard to spot the difference

```
class XYZControllerForEfficientStorageOfStrings  
{  
}
```

```
int a = 1;  
if ( O == 1 )  
    a = 01;  
else  
    l = 01;
```

Difference between O and 0?
Difference between L and 1?
Maybe in another font?

```
int a = 1;  
if ( 0 == 1 )  
    a = 01;  
else  
    l = 01;
```

Avoid disinformation

- Avoid false clues that obscure the meaning of code
- Don't name something a **List** if it is not a list
 - better names: **accountGroup** or just simply **accounts**
- Don't use names which vary in small ways
 - it takes time to spot the difference
 - it can easily lead to bugs, if we use the wrong one
- Don't use '**0**' and '**1**' as names, they can be easily confused with **0** and **1**

Solution

```
string example;

Set<Account> accounts;

class StringCache
{
}

class StringDatabase
{
}

int node = leaf;
if (current == leaf)
    node = right;
else
    leaf = 1;
```

Problem

```
public static void CopyChars(char a1[], char a2[]) {  
    for (int i = 0; i < a1.Length; i++) {  
        a2[i] = a1[i];  
    }  
}
```

Which is the source, which is the destination?
Cannot be seen from the outside!

```
GetActiveAccount();  
GetActiveAccounts();  
GetActiveAccountInfo();
```

How are the programmers supposed to know
which of these functions to call?

Make meaningful distinctions

- Don't use names like **a1, a2, ..., aN**
 - these names are non-informative
- Don't use noise words like '**a**', '**the**', '**variable**', '**info**', etc.
 - **theMessage** vs. **Message**
 - **NameString** vs. **Name**
 - **AccountData** vs. **Account**
 - **CustomerObject** vs. **Customer**
- Don't reuse variables for different purposes
 - the compiler is *very* clever:
 - it can assign the same register to different variables
 - it can optimize a lot of things (e.g. loops, tail recursion, etc.)
 - make the code readable and leave the rest to the compiler

Solution

```
public static void CopyChars(char source[], char destination[]) {  
    for (int i = 0; i < source.Length; i++) {  
        destination[i] = source[i];  
    }  
}  
  
GetActiveAccount();  
GetAllActiveAccounts();  
GetActiveAccountProfile();
```

Problem

```
class DtaRcrd102 {  
    private DateTime genymdhms;  
    private DateTime modymdhms;  
    private const string pszqint = "102";  
    /* ... */  
};
```

How do you pronounce these?
How can you talk about these?
How do you search these?

```
public class Part {  
    private string m_dsc; // The textual description  
    void SetName(string name) {  
        m_dsc = name;  
    }  
}
```

Use pronounceable names

- Use names that can be pronounced
 - so that you can talk about it
 - “gen why emm dee aich emm ess” vs. `generationTimestamp`
- Examples:
 - “Hey, Mikey, take a look at this record! The `gen why emm dee aich emm ess` is set to tomorrow’s date! How can that be?”
 - vs.
 - “Hey, Mikey, take a look at this record! The `generation timestamp` is set to tomorrow’s date! How can that be?”

Use searchable names

- Use names that can be searched
 - **ymdhms** vs. **timestamp**
 - **10** vs. **MaxClassesPerStudent**
 - a searchable name is better than a constant
- Don't use abbreviations or made up acronyms
 - **ymdhms** vs. **timestamp**
 - Elon Musk also banned making up acronyms at SpaceX:
 - “The key test for an acronym is to ask whether it helps or hurts communication.”
 - commonly known acronyms are fine (**XML**, **GUI**, etc.)

Avoid encodings

- Don't use Hungarian notation (= type is part of the variable name)
 - modern compilers enforce types
 - What if the type of a variable changes? Will you rename it? What if you forget to?
- Don't use member prefixes (`m_name`, `_name`, etc.)
 - classes should be small enough that you do not need them
 - IDEs can color members and local variables differently
 - or just simply write the `this` keyword explicitly
 - people tend to get used to prefixes and get 'prefix-blind'
- Naming interfaces and their implementing classes
 - `IList` vs. `List`
 - `List` vs. `ListImpl` or `CList`
- Encodings make code harder to read
- Exceptions:
 - framework convention (e.g. .NET interfaces are prefixed: `IList`)
 - company coding standard requires it
 - (but think about this rule when creating the coding standard)

Solution

```
class DtaRcrd102 {  
    private DateTime genymdhms;  
    private DateTime modymdhms;  
    private const string pszqint = "102";  
    /* ... */  
};
```



```
class Customer {  
    private DateTime generationTimestamp;  
    private DateTime modificationTimestamp;  
    private const string recordId = "102";  
    /* ... */  
};
```

Solution

```
public class Part {  
    private string m_dsc; // The textual description  
    void SetName(string name) {  
        m_dsc = name;  
    }  
}
```



```
public class Part {  
    private string description;  
    void SetDescription(string description) {  
        this.description = description;  
    }  
}
```

Problem

```
public Account A(List<Account> al, string n)
{
    for (int loopVariable = 0; loopVariable < al.Count; ++loopVariable)
    {
        if (al[loopVariable].Number == n) return al[loopVariable];
    }
    return null;
}

protected string GenerateSqlStatementFromHardCodedValuesAndSafeDataTypes(
{
    StringBuilder sb = new StringBuilder(1024);
    sb.AppendFormat("select comment_id, comment, commentor_id from {0} ",
                   TableName);
    sb.AppendFormat("where Comment_id={0} ", Filter.Value);
    return sb.ToString();
}
```

What does the function do?
What are the parameters?

A long loop variable name
is inconvenient to use.

Too long name.

Length of names

- Shorter names are generally better than longer ones, so long as they are clear
 - encodings, abbreviations and made up acronyms are **not** clear
- If you cannot find a clear short name, use a longer name
 - don't be afraid to make a name long
 - a long descriptive name is better than a short enigmatic name
 - a long descriptive name is better than a long descriptive comment
 - use a naming convention that allows multiple words to be easily read
- Single-letter names can **only** be used as local variables inside short methods
 - e.g. loop variables can be **i**, **j**, **k** (but not **l**)
- The length of a name should correspond to the size of its scope
 - local variables can be short
 - method names, parameter names and class names should be longer

Solution

We know what the function does.

```
public Account FindAccountByName(List<Account> accounts, string name)
```

We know what the parameters are.

```
{  
    for (int i = 0; i < accounts.Count; ++i)  
    {  
        if (accounts[i].Name == name) return accounts[i];  
    }  
    return null;  
}
```

Short and meaningful name.

Short loop variable is more convenient.

```
protected string GenerateSafeSql()  
{  
    StringBuilder sb = new StringBuilder(1024);  
    sb.AppendFormat("select comment_id, comment, commentator_id from {0} ",  
                  TableName);  
    sb.AppendFormat("where Comment_id={0} ", Filter.Value);  
    return sb.ToString();  
}
```

Class names

- Classes and objects should have noun or noun phrase names like **Customer**, **WikiPage**, **Account**, and **AddressParser**
- Avoid words like **Manager**, **Processor**, **Data**, or **Info** in the name of a class
- A class name should not be a verb
- Exceptions:
 - extracting behavior, e.g. Visitor, Strategy design patterns

Method names

- Methods should have verb or verb phrase names like `PostPayment`, `DeletePage`, or `Save`
- Accessors, mutators, and predicates should be named for their value and prefixed with `Get`, `Set`, and `Is`

```
string name = employee.Name;  
customer.Name = "Mike";  
if (paycheck.IsPosted)...
```

```
String name = employee.getName();  
customer.setName("Mike");  
if (paycheck.isPosted())...
```

- When constructors are overloaded, use static factory methods with names that describe the arguments
 - consider enforcing their use by making the corresponding constructors private

```
Complex centerPoint = Complex.FromRealNumber(23.0);
```

- vs.

```
Complex centerPoint = new Complex(23.0);
```

Problem

```
class Bank
{
    Set<Account> FetchAccounts();
}

class Account
{
    Set<Person> RetrieveOwners();
}

class Person
{
    string GetName();
}
```

Inconsistent names

Pick one word per concept

- Don't use different names for the same concept
- They are confusing
- They are hard to remember
- Examples:
 - **fetch, retrieve, get**
 - **stop, kill, quit, end**
 - **controller, manager, driver**
 - What is the essential difference between a DeviceManager and a ProtocolController? Why are both not controllers or both not managers? Are they both Drivers really?

Solution

```
class Bank
{
    Set<Account> GetAccounts();
}

class Account
{
    Set<Person> GetOwners();
}

class Person
{
    string GetName();
}
```

Use the right domain

- Use solution domain names
 - code is read by programmers, they know computer science
 - use the appropriate algorithm names, pattern names, math terms, etc.
 - e.g. `AccountVisitor`, `JobQueue`
- Use problem domain names
 - it is easier to make changes and extend the model if the requirements change
- Separating solution and problem domain concepts is part of the job of a good programmer and designer
 - the code that has more to do with problem domain concepts should have names drawn from the problem domain

Problem

```
private void PrintGuessStatistics(char candidate, int count) {  
    string number;  
    string verb;  
    string pluralModifier;  
    if (count == 0) {  
        number = "no";  
        verb = "are";  
        pluralModifier = "s";  
    } else if (count == 1) {  
        number = "1";  
        verb = "is";  
        pluralModifier = "";  
    } else {  
        number = count.ToString();  
        verb = "are";  
        pluralModifier = "s";  
    }  
    String guessMessage = string.Format(  
        "There {0} {1} {2}{3}", verb, number, candidate, pluralModifier  
    );  
    Print(guessMessage);  
}
```

We have to read through and understand the whole function to know the context and meaning of the variables.



Add meaningful context

- There are a few names which are meaningful in and of themselves, but most are not
 - Example:
 - If we see `firstName`, `lastName`, `street`, `houseNumber`, `city`, `state`, and `zipcode`, we can infer that they are part of an `Address`
 - But what if you just saw the `state` variable being used alone in a method? Would you automatically infer that it was part of an address?
- Place names in context for your reader by enclosing them in well-named classes, functions, or namespaces
 - e.g. `Address` class
- When all else fails, then prefixing the name may be necessary as a last resort
 - e.g. `addrFirstName`, `addrLastName`, `addrState`

Don't add meaningless context

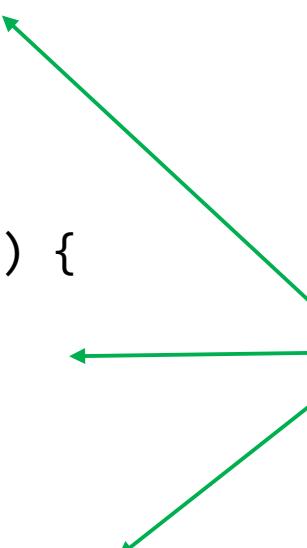
- Don't prefix every class with the same thing
 - e.g. **GSDAccountAddress**, **GSDPerson**
 - the prefix won't add any meaningful context to the class name
 - the IDE will list all the classes when you start typing
- Examples:
 - **accountAddress** and **customerAddress** are fine names for instances of the class **Address**
 - **accountAddress** and **customerAddress** are poor names for classes
 - **MAC** vs. **MACAddress**
 - **URI** vs. **WebAddress**

Solution

```
public class GuessStatisticsMessage {  
    private string number;           ← A separate class with a meaningful name  
    private string verb;            provides a clear context for all the variables.  
    private string pluralModifier;  
  
    public string Make(char candidate, int count) {  
        CreatePluralDependentMessageParts(count);  
        return string.Format("There {0} {1} {2}{3}",  
                             verb, number, candidate, pluralModifier );  
    }  
  
    private void CreatePluralDependentMessageParts(int count) {  
        if (count == 0) {  
            ThereAreNoLetters();          ← Smaller functions are easier  
        } else if (count == 1) {         to read and understand.  
            ThereIsOneLetter();  
        } else {  
            ThereAreManyLetters(count);  ← Cases are separated into  
        }                                smaller functions with  
    }                                    meaningful names.  
...  
}
```

Solution

```
public class GuessStatisticsMessage {  
    ...  
  
    private void ThereAreManyLetters(int count) {  
        number = count.ToString();  
        verb = "are";  
        pluralModifier = "s";  
    }  
  
    private void ThereIsOneLetter() {  
        number = "1";  
        verb = "is";  
        pluralModifier = "";  
    }  
  
    private void ThereAreNoLetters() {  
        number = "no";  
        verb = "are";  
        pluralModifier = "s";  
    }  
}
```



Cases are separated into smaller functions with meaningful names.

Smaller functions are easier to read and understand.

Functions

Problem

```
public static string TestableHtml(PageData pageData, boolean includeSuiteSetup) {  
    WikiPage wikiPage = pageData.GetWikiPage();  
    StringBuilder builder = new StringBuilder();  
    if (pageData.HasAttribute("Test")) {  
        if (includeSuiteSetup) {  
            WikiPage suiteSetup = PageCrawlerImpl.GetInheritedPage(SuiteResponder.SuiteSetupName, wikiPage);  
            if (suiteSetup != null) {  
                WikiPagePath pagePath = suiteSetup.GetPageCrawler().GetFullPath(suiteSetup);  
                String pagePathName = PathParser.Render(pagePath);  
                builder.Append("!include -setup .").Append(pagePathName).Append("\n");  
            }  
        }  
        WikiPage setup = PageCrawlerImpl.GetInheritedPage("SetUp", wikiPage);  
        if (setup != null) {  
            WikiPagePath setupPath = wikiPage.GetPageCrawler().GetFullPath(setup);  
            String setupPathName = PathParser.Render(setupPath);  
            builder.Append("!include -setup .").Append(setupPathName).Append("\n");  
        }  
        builder.Append(pageData.GetContent());  
        if (pageData.HasAttribute("Test")) {  
            WikiPage teardown = PageCrawlerImpl.GetInheritedPage("TearDown", wikiPage);  
            if (teardown != null) {  
                WikiPagePath tearDownPath = wikiPage.GetPageCrawler().GetFullPath(teardown);  
                String tearDownPathName = PathParser.Render(tearDownPath);  
                builder.Append("\n").Append("!include -teardown .").Append(tearDownPathName).Append("\n");  
            }  
            if (includeSuiteSetup) {  
                WikiPage suiteTeardown = PageCrawlerImpl.GetInheritedPage(SuiteResponder.SuiteTearDownName, wikiPage);  
                if (suiteTeardown != null) {  
                    WikiPagePath pagePath = suiteTeardown.GetPageCrawler().GetFullPath(suiteTeardown);  
                    String pagePathName = PathParser.Render(pagePath);  
                    builder.Append("!include -teardown .").Append(pagePathName).Append("\n");  
                }  
            }  
        }  
        pageData.SetContent(builder.ToString());  
        return pageData.GetHtml();  
    }  
}
```

Function is too large!

Problem

High abstraction level: WikiPage

```
public static string TestableHtml(PageData pageData, boolean includeSuiteSetup) {
    WikiPage wikiPage = pageData.GetWikiPage();
    StringBuilder builder = new StringBuilder();
    if (pageData.HasAttribute("Test")) {
        if (includeSuiteSetup) {
            WikiPage suiteSetup = PageCrawlerImpl.GetInheritedPage(SuiteResponse);
            if (suiteSetup != null) {
                WikiPagePath pagePath = suiteSetup.getPageCrawler().GetFullPath();
                String pagePathName = PathParser.Render(pagePath);
                builder.Append("!include -setup .").Append(pagePathName).Append("\n");
            }
        }
    }
}
```

Deeply nested conditions
(repeated in the last 15 lines!)

Low abstraction level: StringBuilder

Multiple page resolution

And this is just the first 11 lines of 38!

Functions should be small

- Functions should hardly ever be 20 lines long
- Best if functions are 2-4 lines long
- Functions should be easy to read
- Functions should tell a story

Blocks and indenting

- Blocks within if statements, else statements, while statements, try-catch statements, and so on should be one line long
 - probably that line should be a function call
- This also adds documentary value
 - the function called within the block can have a nicely descriptive name
- Functions should not be large enough to hold nested structures
 - the indent level of a function should not be greater than one or two
- This makes the functions easier to read and understand

Functions should do one thing

- *Functions should do one thing. They should do it well. They should do it only.*
- What is one thing?
 - If a function does only those steps that are one level below the stated name of the function, then the function is doing one thing.
 - Exception handling, logging, concurrency, etc. are already one abstraction level, they are already one thing!
- What is more than one thing?
 - If you can extract another function from it with a name that is not merely a restatement of its implementation
 - If the function can be divided into sections
 - If there are multiple abstraction levels within the function

One abstraction level per function

- The statements within a function should all be at the same level of abstraction
- Mixing levels of abstraction within a function is always confusing
- Readers may not be able to tell whether a particular expression is an essential concept or a detail
- Once details are mixed with essential concepts, more and more details tend to accrete within the function

Stepdown rule

- We want the code to read like a top-down narrative
- We want to be able to read the program as though it were a set of TO paragraphs:
 - *To include the setups and teardowns, we include setups, then we include the test page content, and then we include the teardowns.*
 - *To include the setups, we include the suite setup if this is a suite, then we include the regular setup.*
 - *To include the suite setup, we search the parent hierarchy for the "SuiteSetUp" page and add an include statement with the path of that page.*
 - *To search the parent ...*
- Order of the functions:
 - every function should be followed by those at the next level of abstraction
 - so that we can read the program, descending one level of abstraction at a time, as we read down the list of functions
- It is an effective technique for keeping the abstraction level consistent

Solution

```
public class SetupTeardownIncluder {  
    private PageData pageData;  
    private bool isSuite;  
    private WikiPage testPage;  
    private StringBuilder nextPageContent;  
    private PageCrawler pageCrawler;  
  
    public static string Render(PageData pageData) {  
        return Render(pageData, false);  
    }  
  
    public static string Render(PageData pageData, bool isSuite)  
    {  
        return new SetupTeardownIncluder(pageData).Render(isSuite);  
    }  
  
    private SetupTeardownIncluder(PageData pageData) {  
        this.pageData = pageData;  
        testPage = pageData.GetWikiPage();  
        pageCrawler = testPage.GetPageCrawler();  
        nextPageContent = new StringBuilder();  
    }  
}
```

Function lengths: 2-4 lines

Static factory methods for different cases

Private constructor

Solution

```
public class SetupTeardownIncluder {  
    ...  
  
    private string Render(bool isSuite) {  
        this.isSuite = isSuite;  
        if (IsTestPage())  
            IncludeSetupAndTeardownPages();  
        return pageData.GetHtml();  
    }  
}
```

Function lengths: 2-4 lines

One line long “if”

```
private boolean IsTestPage() {  
    return pageData.hasAttribute("Test");  
}
```

Stepdown rule:
lowering abstraction levels

```
private void IncludeSetupAndTeardownPages() {  
    IncludeSetupPages();  
    IncludePageContent();  
    IncludeTeardownPages();  
    UpdatePageContent();  
}  
...  
}
```

Stepdown rule:
lowering abstraction levels

Solution

```
public class SetupTeardownIncluder {  
    ...  
    private void IncludePageContent() {  
        newPageContent.Append(pageData.GetContent());  
    }  
  
    private void IncludeTeardownPages() {  
        IncludeTeardownPage();  
        if (isSuite)  
            IncludeSuiteTeardownPage(); ← One line long “if”  
    }  
  
    private void IncludeTeardownPage() {  
        Include("TearDown", "-teardown");  
    }  
  
    private void IncludeSuiteTeardownPage() {  
        Include(SuiteResponder.SuiteTearDownName, "-teardown");  
    }  
  
    private void UpdatePageContent() {  
        pageData.SetContent(newPageContent.ToString());  
    }  
}
```

Function lengths: 2-4 lines

Stepdown rule:
lowering abstraction levels

Solution

```
public class SetupTeardownIncluder {  
    ...  
    private void Include(string pageName, string arg) {  
        WikiPage inheritedPage = FindInheritedPage(pageName);  
        if (inheritedPage != null) {  
            string pagePathName = GetPathNameForPage(inheritedPage);  
            BuildIncludeDirective(pagePathName, arg);  
        }  
    }  
    private WikiPage FindInheritedPage(string pageName) {  
        return PageCrawlerImpl.GetInheritedPage(pageName, testPage);  
    }  
    private string GetPathNameForPage(WikiPage page) {  
        WikiPagePath pagePath = pageCrawler.GetFullPath(page);  
        return PathParser.Render(pagePath);  
    }  
    private void BuildIncludeDirective(string pagePathName, string arg) {  
        newPageContent.Append("\n!Include ").Append(arg).Append(" .").Append()  
    }  
}
```

Function lengths: 2-4 lines

Stepdown rule:
lowering abstraction levels

Almost one line long “if”

Problem

void Transform(Point p) ← Mutation: transform in place

string a = ...

string b = ...

AssertEquals(a, b); ←

Which is the expected value?
Which is the actual value?

Save("Document.txt", false, true, false, false)

What do all these options mean?

Function arguments

- Functions should have as few arguments as possible
 - the ideal number of arguments for a function is zero
 - next comes one
 - followed closely by two
 - three arguments should be avoided where possible
- More than three arguments requires very special justification
- Too many arguments usually means that abstraction levels are mixed
- Arguments are even harder from a testing point of view:
testing all the combinations

One argument

- Two common reasons for supplying a single argument:
 - asking a question about that argument
 - operating on that argument, transforming it into something else and *returning it*
- Functions with single arguments may be events
- Passing a boolean value (flag) into a function is ugly
 - it complicates the signature of the method
 - indicating that the function does more than one thing
- Examples:
 - `boolean FileExists("MyFile")` – question
 - `InputStream FileOpen("MyFile")` – transformation
 - `void IncludeSetupPageInto(StringBuffer pageText)` – mutation
 - should be: `void IncludeSetupPage()`, and use 'pageText' as a field
 - `void Transform(StringBuffer out)` – mutation
 - should be: `StringBuffer Transform(StringBuffer in)`, even if it just returns 'in'
 - `void Render(bool isSuite)` – flag
 - should be: `void RenderForSuite()` and `void RenderForSingleTest()`

Two arguments

- A function with two arguments is harder to understand than a function with one argument
- Two arguments are appropriate when they are ordered components of a single value
- Two arguments are problematic if they have no natural ordering
- Two arguments aren't evil, but they come at a cost
 - consider turning them into single arguments
 - e.g. making one of them a field
- Examples:
 - `new Point(0,0)` – OK: ordered components of a single value
 - `AssertEquals(expected, actual)` – no natural ordering: requires practice to learn
 - should be: `AssertExpectedEqualsActual(expected, actual)`
 - `WriteField(outputStream, name)` – probably OK
 - better: `WriteField(name)` and making 'outputStream' a field
 - `string.Format(string format, params object[] args)` – OK, still counts as two arguments, since the items of args are treated identically, as if args were a normal array or list

Three or more arguments

- Functions that take three arguments are significantly harder to understand than functions with fewer arguments
- There are issues of
 - ordering: which parameter is which
 - pausing: thinking about the meaning of each parameter
 - ignoring: we have to include common parameters (e.g. stream)
 - testing: testing all the combinations of values is hard
- Consider wrapping the arguments into a class of their own
 - they probably deserve a name of their own
- Examples:
 - **AssertEquals(message, expected, actual)** –
Can you remember which is which?
What if you extend an **AssertEquals(expected, actual)** call?
 - **AssertEquals(1.0, amount, 0.001)** –
Probably worth the thinking: equality of floating point values
 - **Save("Document.txt", false, true, false, false)** –
What do all these options mean?
Should be: **Save("Document.txt", saveOptions)**

Naming functions

- Functions should have zero, one or two arguments
- The name can explain the intent and the order of the arguments
- Name of a function with zero arguments:
 - a verb can be nice, but can be longer
 - e.g. `Save()`, `Quit()`, `Transform()`, `IncludePage()`, `UpdatePageContent()`
- Name of a function with one argument:
 - a verb-noun pair can be nice by encoding the name of the argument into the function name
 - e.g. `WriteField(name)` instead of just `Write(name)`
- Name of a function with two arguments:
 - encode the names of the arguments into the function name
 - e.g. `AssertExpectedEqualsActual(expected, actual)` instead of just `AssertEquals(expected, actual)`

Solution

Point Transform(Point p) ← Transformed point as a result

string a = ...

string b = ...

Assert.AreEqual(a, b);

We know which is which.

var saveOptions = new SaveOptions();

saveOptions.AddUtf8Bom = true; ← We know what the options mean.

Save("Document.txt", saveOptions)

Problem

```
public class UserValidator {  
    private Cryptographer cryptographer; Query function  
  
    public bool CheckPassword(string userName, string password) {  
        User user = UserGateway.FindByName(userName);  
        if (user != User.None) {  
            string codedPhrase = user.GetPhraseEncodedByPassword();  
            string phrase = cryptographer.Decrypt(codedPhrase, password);  
            if (phrase == "Valid Password") {  
                Session.Initialize(); Side effect  
                return true;  
            }  
        }  
        return false;  
    }  
}
```

```
if (Set("username", "alice")) { Query and command at the same time  
    ...  
}
```

Have no side effects

- Your function promises to do one thing
- But with side effects it also does other hidden things
 - unexpected changes to the variables of its own class
 - changes to the parameters passed
 - changes to system globals
- Side effects can create temporal coupling
 - the function can only be called at certain times
- Side effects can also cause concurrency issues

Command and query separation

- Functions should either do something or answer something, but not both
 - either change the state of an object
 - or return some information about that object
- Doing both often leads to confusion
- Exceptions:
 - atomic operations for thread-safety
 - e.g. `TestAndSet(flag)`, `Interlocked.CompareExchange()`

Solution

```
public class UserValidator {  
    private Cryptographer cryptographer; Query function  
  
    public bool CheckPassword(string userName, string password) {  
        User user = UserGateway.FindByName(userName);  
        if (user != User.None) {  
            string codedPhrase = user.GetPhraseEncodedByPassword();  
            string phrase = cryptographer.Decrypt(codedPhrase, password);  
            if (phrase == "Valid Password") {  
                Session.Initialize(); Remove this line, do it somewhere else.  
                return true;  
            }  
        }  
        return false;  
    }  
}
```

```
if (AttributeExists("username")) {  
    SetAttribute("username", "alice"); Command  
}
```

How to write functions like this?

- It is usually easier to write longer functions
 - with a lot of indenting, loops, conditions, etc.
 - with a lot of arguments
 - with duplicated code
- You should have unit tests to cover all the lines of these functions
- Then you can refine the code while keeping the tests passing
 - extracting new functions or new classes
 - changing names
 - eliminating duplication
 - shrinking methods
 - reordering methods

Writing and omitting comments

Comments

- Proper use of comments is to compensate for our *failure to express ourselves in code*
- Well-placed comments can be very useful
- But meaningless or even lying comments can be very damaging
 - code changes and evolves, chunks are moving around
 - comments don't always follow the change
- Inaccurate comments are far worse than no comments at all
 - they are misleading
 - they contain old rules that need not or should not be followed any longer
- The only source of accurate information is the *code*
- Don't comment bad code – rewrite it!

Problem

```
public void LoadProperties()
{
    try
    {
        string propertiesPath =
            Path.Combine(propertiesLocation, PropertyFileName);
        StreamReader propertiesStream = new StreamReader(propertiesPath);
        loadedProperties.Load(propertiesStream);
    }
    catch(IOException e)
    {
        // No properties files means all defaults are loaded
    }
}
```

What does this comment mean?

Apparently, if there is an IOException, then there is no properties file.

But who loads the defaults? Are they already loaded?

Or loadedProperties.Load() loads them?

Or they should have been loaded in the catch block, but have been forgotten?

Bad comments: Mumbling

- Don't just write a comment because you feel you should or it is required
- If you decide to write a comment make sure it is the best comment you can write
- The comment should be meaningful on its own within its context
 - any comment that forces you to look in another module for the meaning of that comment has failed to communicate to you

Problem

The comment just repeats what the code does. It is redundant.



```
// Utility method that returns when this.closed is true.  
// Throws an exception if the timeout is reached.  
public void WaitForClose(long timeoutMillis)  
{  
    if (!closed)  
    {  
        Wait(timeoutMillis);  
        if (!closed)  
            throw new Exception("Sender could not be closed");  
    }  
}
```

Bad comments: Redundant comments

- Redundant comment:
 - it is not more informative than the code
 - it is not easier to read than the code
 - it may be less precise than the code and may mislead the reader
- Don't write redundant comment
- Instead:
 - write comments that justify the code, or provide intent or rationale
 - or just omit the comment if the code explains itself

Problem

The comment is also subtly misleading:

The method does not return *when* `this.closed` becomes true.

It returns *if* `this.closed` is true; otherwise, it waits for a time-out and then throws an exception if `this.closed` is still not true.



```
// Utility method that returns when this.closed is true.  
// Throws an exception if the timeout is reached.  
public void WaitForClose(long timeoutMillis)  
{  
    if (!closed)  
    {  
        Wait(timeoutMillis);  
        if (!closed)  
            throw new Exception("Sender could not be closed");  
    }  
}
```

Bad comments: Misleading comments

- Sometimes a comment written with all the best intentions may be inaccurate
 - it does not explain precisely what the code does
 - so it can be slightly misleading
 - another programmer calls the function based on the comment and later he can debug why it behaves differently than expected
- Don't write redundant comment, it can be misleading
- Only the code is accurate enough

Problem

Probably mandatory comment with no added value.



```
/// <summary>
/// Adds a CD
/// </summary>
/// <param name="title">The title of the CD</param>
/// <param name="author">The author of the CD</param>
/// <param name="tracks">The number of tracks on the CD</param>
/// <param name="durationInMinutes">The duration of the CD in minutes</param>
public void AddCD(string title, string author,
    int tracks, int durationInMinutes)
{
    CD cd = new CD();
    cd.Title = title;
    cd.Author = author;
    cd.Tracks = tracks;
    cd.Duration = duration;
    cdList.Add(cd);
}
```

Bad comments: Mandatory comments

- It is a bad idea to have rules like:
 - every function must have a documentation comment
 - every variable must have a comment
- Comments like this just clutter up the code, propagate lies, and lend to general confusion
- Don't make comments mandatory for everything

Problem

```
public class AnnualDateRule
{
    /// <summary>
    /// Default constructor ← No, really?
    /// </summary>
    protected AnnualDateRule()
    {
    }

    /// <summary>
    /// The day of the month. ← I would never have guessed...
    /// </summary>
    private int dayOfMonth;

    /// <summary>
    /// Returns the day of the month. ← Just noise...
    /// </summary>
    /// <returns>the day of the month</returns>
    public int DayOfMonth
    {
        get { return dayOfMonth; }
    }
}
```

Problem

```
// The name. ← Noise...
private string name;

// The version.
private string version;

// The licenceName.
private string licenceName;

// The version. ← Scary noise!!!
private string info;
```

Bad comments: Noise comments

- Noise comments just restate the obvious and provide no new information
- These comments are so noisy that we learn to ignore them
- Eventually they begin to lie as the code around them changes
- Scary noise:
 - If authors aren't paying attention when comments are written (or pasted), why should readers be expected to profit from them?
- Don't write noise comments, especially if the code explains itself

Problem

```
private void StartSending()
{
    try
    {
        DoSending();
    }
    catch (SocketException e)
    {
        // normal. someone stopped the request. OK
    }
    catch (Exception e)
    {
        try
        {
            response.Add(ErrorResponder.MakeExceptionString(e));
            response.CloseAll();
        }
        catch (Exception e1)
        {
            //Give me a break! The developer was angry.
        }
    }
}
```

Bad comments: Angry comments

- Angry comments are also just noise, they provide no new information
- Frustration can be resolved by improving the structure of the code
- Replace the temptation to create noise with the determination to clean your code

Problem

Could we rewrite the code so that we can omit the comment?



```
// does the module from the global list <mod> depend on the  
// subsystem we are part of?  
if (smodule.GetDependSubsystems().Contains(subSysMod.GetSubSystem()))
```

Bad comments: Comment instead of a function or variable

- Don't write a comment if you can express the same thing in code
- Write expressive code instead

Solution

```
List moduleDependees = smodule.GetDependSubsystems();  
string ourSubSystem = subSysMod.GetSubSystem();  
if (moduleDependees.Contains(ourSubSystem))
```



The code is easy to understand, no need for comments

Problem

```
public class ArrayBuilder<T>
{
    // Fields //////////////////////////////// ↑

    private int count;
    private T[] items;

    // Constructors //////////////////////////////// ↑ Noise ←

    public ArrayBuilder()
    { ... }

    // Properties //////////////////////////////// ↑

    public int Count { get { return this.count; } }

    // Methods //////////////////////////////// ↑

    public void Add(T item)
    { ... }

    public void AddRange(IEnumerable<T> items)
    { ... }
}
```

The diagram shows a red arrow pointing from the word 'Noise' at the end of the second line to the first section marker ('// Fields'). Another red arrow points from 'Noise' to the second section marker ('// Constructors'). A third red arrow points from 'Noise' to the third section marker ('// Properties').

Bad comments: Banner comments

- Banner or position marker comments add noise to the code
- If you overuse banners, they'll fall into the background noise and be ignored
- Use them very sparingly, and only when the benefit is significant

Problem

```
static void Main(string[] args)
{
    string line;
    int lineCount = 0;
    int charCount = 0;
    int wordCount = 0;
    try
    {
        while ((line = Console.ReadLine()) != null)
        {
            lineCount++;
            charCount += line.Length;
            string[] words = line.Split(' ', '\t');
            wordCount += words.Length;
        } //while ← Closing while
        Console.WriteLine("wordCount = " + wordCount);
        Console.WriteLine("lineCount = " + lineCount);
        Console.WriteLine("charCount = " + charCount);
    } // try ← Closing try
    catch (IOException e)
    {
        Console.Error.WriteLine("Error:" + e.Message);
    } //catch ← Closing catch
} //main ← Closing main
```

Bad comments: Closing brace comments

- Closing brace comments might make sense for long functions with deeply nested structures
- But functions should not be long and have deeply nested structures
- Make sure to format the code properly
- If you find yourself wanting to mark your closing braces, try to shorten your functions instead

Problem

```
InputStreamResponse response = new InputStreamResponse();
response.SetBody(formatter.GetResultStream(), formatter.GetByteCount());
// InputStream resultsStream = formatter.GetResultStream();
// StreamReader reader = new StreamReader(resultsStream);
// response.SetContent(reader.Read(formatter.GetByteCount()));
```



Commented-out code: why is it there?

Bad comments: Commented-out code

- Commented-out code always seems important
 - Are they reminders?
 - Do they indicate an imminent change?
 - Were they just left there years ago and have been forgotten?
- Others who see it won't have the courage to delete it
- So commented-out code just gathers throughout the code
- Don't leave commented-out code in the source
- Use a source control system
 - you can safely delete the code
 - and you can restore it anytime

Problem

- * Changes (from 11-Oct-2001)
- * -----
- * 11-Oct-2001 : Re-organised the class and moved it to new package
- * com.jrefinery.date (DG);
- * 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
- * class (DG);
- * 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
- * class is gone (DG); Changed getPreviousDayOfWeek(),
- * getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
- * bugs (DG);
- * 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
- * 29-May-2002 : Moved the month constants into a separate interface
- * (MonthConstants) (DG);
- * 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
- * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
- * 13-Mar-2003 : Implemented Serializable (DG);
- * 29-May-2003 : Fixed bug in addMonths method (DG);
- * 04-Sep-2003 : Implemented Comparable. Updated the isInRange javadocs (DG);
- * 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);



Journal comment: why is it there?

Bad comments: Journal comments

- Long ago there was a good reason to create and maintain these log entries at the start of every file
- But today we have source control systems that maintain these entries
- Don't write journal comments, they just add more noise to the code

Problem

```
/* Added by Rick */ ← Byline comment: why is it there?  
public class Date  
{  
}
```

Bad comments: Attributions and bylines

- You might think that such comments would be useful in order to help others know who to talk to about the code
- But source control systems are much better at remembering who added what, when
- There is no need to pollute the code with little bylines

Problem

```
/**  
 * Task to run fit tests.  
 * This task runs fitness tests and publishes the results.  
 * <p/>  
 * <pre>  
 * Usage:  
 * <taskdef name="execute-fitness-tests"  
 * classname="fitness.ant.ExecuteFitnessTestsTask";  
 * classpathref="classpath" />  
 * OR  
 * <taskdef classpathref="classpath";  
 * resource="tasks.properties" />  
 * <p/>  
 * <execute-fitness-tests  
 * suitepage="FitNesse.SuiteAcceptanceTests";  
 * fitnesseport="8082";  
 * resultsdir="${results.dir}";  
 * resultshtmllpage="fit-results.html";  
 * classpathref="classpath" />  
 * </pre>  
 */
```

Hard to read

Bad comments: HTML comments

- HTML makes the comments hard to read in the one place where they should be easy to read: the editor/IDE
- It should be the responsibility of the documentation generator tool, and not the programmer to add HTML markers
- Unfortunately, Visual Studio prefers HTML comments

Problem

Nonlocal information.
Redundant.
The function has no control over it.
Will it remain accurate?

```
/// <summary>
/// Port on which fitness would run. Defaults to <b>8082</b>.
/// </summary>
/// <param name="fitnessPort"></param>
public void SetFitnessPort(int fitnessPort)
{
    this.fitnessPort = fitnessPort;
}
```



Bad comments: Nonlocal Information

- Don't offer systemwide information in the context of a local comment
 - it is redundant
 - it is not describing the function, but some other, far distant part of the system
 - there is no guarantee that this comment will be changed when the systemwide information is changed
- If you must write a comment, then make sure it describes the code it appears near

Problem

```
/*  
Extensible Markup Language (XML) is a simple, very flexible  
text format derived from SGML (ISO 8879). Originally designed  
to meet the challenges of large-scale electronic publishing,  
XML is also playing an increasingly important role in the  
exchange of a wide variety of data on the Web and elsewhere.
```

XML documents are made up of storage units called entities, which contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form markup. Markup encodes a description of the document's storage layout and logical structure. XML provides a mechanism to impose constraints on the storage layout and logical structure.

```
*/  
public string ToXml()  
{  
...  
}
```

Too much and irrelevant information

Bad comments: Too much information

- Don't put interesting historical discussions or irrelevant descriptions of details into your comments
- Include only the relevant details
- Include a reference to the standard, if necessary
- The details of the standard are usually relevant only if you want to directly implement and use the standard
 - exchanging XML files is indirect use

Problem

```
/*
 * start with an array that is big enough to hold all the pixels
 * (plus filter bytes), and an extra 200 bytes for header info
 */
this.pngBytes = new byte[((this.width + 1) * this.height * 3) + 200];
```

Why 200?



Why +1?



Why *3?

Does this relate to +1? Or *3? Or both?

Bad comments: Inobvious connection

- The connection between a comment and the code it describes should be obvious
- The purpose of a comment is to explain code that does not explain itself
- It is bad when a comment needs its own explanation

Good comments: Informative comments

- It is sometimes useful to provide basic information with a comment
- Example:

```
// Returns an instance of the Responder being tested.  
protected abstract Responder responderInstance();
```

- but it is usually better to use the name of the function to convey the information where possible:

```
protected abstract Responder responderBeingTested();
```

- A better example:

```
// format matched kk:mm:ss EEE, MMM dd, yyyy  
Pattern timeMatcher = Pattern.compile(  
    "\\\d*:\\\\d*:\\\\d* \\\\w*, \\\\w* \\\\d*, \\\\d*");
```

- easier to read than trying to understand the regular expression
- but make sure the comment remains in sync with the expression

Good comments: Explanation of intent

- A comment is valuable if it provides the intent behind a decision
 - it explains why the code works the way it is
 - it helps us to adhere to the original design decisions
- **This is the most important reason to write a comment**
- Example:

```
public void TestConcurrentAddWidgets() {  
    WidgetBuilder widgetBuilder = new WidgetBuilder(typeof(BoldWidget));  
    string text = "'''bold text'''";  
    ParentWidget parent = new BoldWidget(new MockWidgetRoot(), text);  
    FailFlag failFlag = new FailFlag();  
  
    //This is our best attempt to get a race condition  
    //by creating large number of threads.  
    for (int i = 0; i < 25000; i++) {  
        WidgetBuilderThread widgetBuilderThread =  
            new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);  
        Thread thread = new Thread(widgetBuilderThread);  
        thread.Start();  
    }  
    Debug.Assert(!failFlag.IsSet);  
}
```

Good comments: Clarification

- Sometimes it is just helpful to translate the meaning of some obscure code into something that's readable
 - in general it is better to find a way to make the code clear in its own right
 - but when it's part of the standard library, or in code that you cannot alter, then a helpful clarifying comment can be useful
 - make sure the clarification comment is correct
- Examples:

```
Debug.Assert(a.CompareTo(a) == 0); // a == a
Debug.Assert(a.CompareTo(b) != 0); // a != b
Debug.Assert(aa.CompareTo(ab) == -1); // aa < ab
```
- There is a substantial risk, of course, that a clarifying comment is (or later becomes) incorrect
- This explains both why the clarification is necessary and why it's risky

Good comments: Warning of consequences

- Sometimes it is useful to warn other programmers about certain consequences
- Examples:

```
public static SimpleDateFormat MakeStandardHttpDateFormat()
{
    //SimpleDateFormat is not thread safe,
    //so we need to create each instance independently.
    SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z");
    df.setTimeZone(TimeZone.getTimeZone("GMT"));
    return df;
}

// Don't run unless you have some time to kill.
public void _testWithReallyBigFile()
{
    WriteLinesToFile(10000000);
    Debug.Assert(bytesSent > 1000000000);
}
```

Good comments: TODO comments

- TODOs are jobs that the programmer thinks should be done, but for some reason can't do at the moment
- It is sometimes reasonable to leave “To do” notes in the form of `//TODO` comments
- TODO comments are reminders:
 - implement a feature later
 - delete a deprecated feature
 - look at a problem later
 - etc.
- Example:

```
//TODO-MdM these are not needed
// We expect this to go away when we do the checkout model
protected VersionInfo MakeVersion() {
    return null;
}
```
- But TODO is not an excuse to leave bad code in the system!
- Good IDEs provide special features to locate all the TODO comments, so it's not likely that they will get lost

Good comments: Amplification

- A comment may be used to amplify the importance of something that may otherwise seem inconsequential
- Example:

```
string listItemContent = match.Group(3).Trim();
// the trim is real important. It removes the starting
// spaces that could cause the item to be recognized
// as another list.
new ListItemWidget(this, listItemContent, this.level + 1);
return BuildList(text.Substring(match.End()));
```

Good comments: Public API documentation

- A well-described public API is very helpful
- But make sure the comments are meaningful, always up-to-date and not misleading
 - include pre- and post-conditions, exceptions, etc.
 - include samples
- The public API must be documented
 - the user of the library does not see (and does not want to see) your source code
- The inner implementation of the library can follow the clean code principles
 - documentation comments are not generally useful here

Defining and handling exceptions

Problem

```
public int DeleteRegistryPage()
{
    if (DeletePage(page) == E_OK) {
        if (registry.DeleteReference(page.name) == E_OK) {
            if (configKeys.DeleteKey(page.name.MakeKey()) == E_OK){
                logger.log("page deleted");
                return E_OK;
            } else {
                logger.log("ConfigKey not deleted");
            }
        } else {
            logger.log("DeleteReference from registry failed");
        }
    } else {
        logger.log("Delete failed");
    }
    return E_ERROR;
}
```

Application logic

Error handling

Use exceptions rather than error codes

- Using error codes clutters the caller
- Error codes lead to deeply nested if statements, and the caller must deal with the error immediately
 - it is also easy to forget to handle the error
- If you use exceptions then the error processing code can be separated from normal execution
 - the calling code is cleaner
 - its logic is not obscured by error handling
- Functions should do one thing
 - error handing is one thing
 - thus, a function that handles errors should do nothing else
 - extract the bodies of the try and catch blocks out into functions of their own

Error handling

- Error handling is important
- But if it obscures logic, it is wrong
 - too many catch blocks
 - catch blocks containing application logic
- Write try-catch-finally first
 - try defines a scope
 - catch and finally must leave the system in a consistent state
- Define the normal flow in the body of the try
 - don't put application logic in a catch block

Solution

```
public void DeleteRegistryPage(Page page)
{
    try
    {
        TryDeleteRegistryPage(page);
    }
    catch(RegistryException e)
    {
        logger.Log(e);
    }
}

private void TryDeleteRegistryPage(Page page)
{
    DeletePage(page);
    registry.DeleteReference(page.name);
    configKeys.DeleteKey(page.name.MakeKey());
}
```

← Error handling

← Application logic

Provide context with exceptions

- File and location of the error
- Reason of the error
 - convert it to a meaningful error message
 - include the operation that failed
 - type of the failure
- Resolution of the error
 - this helps the programmer to correct the error
 - e.g. what should be configured, where can be more information found, etc.
- Provide enough information for logging to reconstruct the problem if necessary

Problem

```
ACMEPort port = new ACMEPort(12);
try {
    port.open();
} catch (DeviceResponseException e) {
    reportPortError(e);
    logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
    reportPortError(e);
    logger.log("Unlock exception", e);
} catch (GMXError e) {
    reportPortError(e);
    logger.log("Device response exception");
} finally {
    ...
}
```



A lot of catch blocks,
but the same error handling logic

Use unchecked (runtime) exceptions

- Checked exceptions in Java:
 - either must be handled by a catch or must be declared in the method signature
 - the client can reasonably be expected to recover from a checked exception
 - e.g. file not found
- Unchecked (runtime) exceptions in Java:
 - the client cannot do anything to recover from an unchecked exception
 - e.g. division by zero, null pointer
- Problem with checked exceptions:
 - they violate OCP
 - all methods in the call hierarchy must declare the exception
 - any change will result in changing all the methods
 - they violate LSP
 - incompatibility of an interface and its implementation: if an interface does not declare a checked exception that cannot be handled by the implementation of the interface
- Checked exceptions can sometimes be useful if you are writing a critical library: you must catch them
- But in general application development the dependency costs outweigh the benefits
- C# does not have checked exceptions

Define exception classes in terms of a caller's needs

- Classify exceptions based on how they are caught
- Exceptions signal that something went wrong
- The low-level details are unimportant
- Don't force the caller to write many catch-blocks
 - their bodies will be the same
- If an external library forces too many exceptions on you, wrap it with your API
- It is always a good idea to wrap third-party APIs
 - minimizes dependency on it
 - easier to move to another library
 - easier to mock it for testing
 - you are not bound to the vendor's design choices: you can have your own API

Solution

```
LocalPort port = new LocalPort(12);
try {
    port.open();
} catch (PortDeviceFailure e) {
    reportError(e);
    logger.log(e.getMessage(), e);
} finally {
    ...
}
```

Problems

```
public void RegisterItem(Item item) {  
    if (item != null) {  
        ItemRegistry registry = persistentStore.GetItemRegistry();  
        if (registry != null) {  
            Item existing = registry.GetItem(item.getID());  
            if (existing.GetBillingPeriod().HasRetailOwner()) {  
                existing.Register(item);  
            }  
        }  
    }  
}
```

Null check forgotten

Null checks

```
List<Employee> employees = GetEmployees();  
if (employees != null) {  
    foreach (Employee e in employees) {  
        totalPay += e.GetPay();  
    }  
}
```

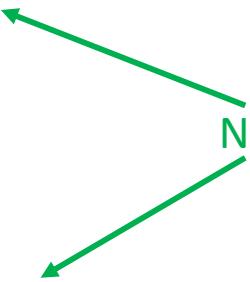
Don't return null

- If something is wrong, throw an exception
- If a collection is to be returned, return an empty collection
- If the method performs some operation, return the neutral element of the operation
- Return a special case object, e.g. NullObject
- The problem of returning null:
 - the client code will be cluttered with null-checks
 - if a null check is missing, there will be a NullReferenceException from very deep

Solution

```
public void RegisterItem(Item item) {  
    if (item == null) throw new ArgumentNullException(nameof(item));  
    ItemRegistry registry = persistentStore.GetItemRegistry();  
    Item existing = registry.GetItem(item.getID());  
    if (existing.GetBillingPeriod().HasRetailOwner()) {  
        existing.Register(item);  
    }  
}
```

No null checks are necessary



```
List<Employee> employees = GetEmployees();  
foreach (Employee e in employees) {  
    totalPay += e.GetPay();  
}
```

Problem

```
public class MetricsCalculator
{
    public double Projection(Point p1, Point p2) {
        return (p2.x - p1.x) * 1.5;
    }
    ...
}
```

```
calculator.Projection(null, new Point(12, 13));
```



Passing null will result in NullReferenceException

Don't pass null

- Don't pass null unless the API expects it
- If the API is not prepared, NullReferenceExceptions will fly
- If you are writing the API check the input parameters, and throw ArgumentNullException
- If the programming language allows it, forbid undesired null values in your API
 - compile time checking
 - (reference types will be non-nullable by default in C# 8)

Good example

```
public class MetricsCalculator
{
    public double Projection(Point p1, Point p2) {
        if (p1 == null) throw new ArgumentNullException(nameof(p1));
        if (p2 == null) throw new ArgumentNullException(nameof(p2));
        return (p2.x - p1.x) * 1.5;
    }
}
```



We will get an ArgumentNullException immediately,
instead of a NullReferenceException later from somewhere deep in the code.

Objects and data structures

Data structures

- Main focus is data
 - data fields
- No behavior
- Typical OO implementation:
 - private fields
 - public properties / getters-setters
 - or just readonly properties / getters if data is immutable
- Behavior in separate functions

Objects

- Main focus is behavior
- Data representation is hidden
- Public interface allows only to manipulate the essence of the data
 - e.g. the x and y coordinates of a point must be set together
- Don't just automatically add getters-setters, think about the abstract representation of the data hidden

Data/object anti-symmetry

Procedural code	Object-oriented code
Code using data structures	Code focusing on behavior
Behavior in separate functions	Data representation hidden
Easy to add new functions without changing existing data structures	Easy to add new classes without changing existing functions
Hard to add new data structures because all the functions must change	Hard to add new functions because all the classes in the hierarchy must change

Data/object anti-symmetry

- Things that are hard for OO are easy for procedures
- Things that are hard for procedures are easy for OO
- In a complex system:
 - there are cases when adding data types is more frequent than adding functions
 - use OO
 - there are cases when adding functions is more frequent than adding data types
 - use procedural code

Law of Demeter (LoD)

- Methods should only be called on objects directly known
- “Talk to friends not to strangers”
- “Train wreck”:
 - `ctx.GetOptions().GetTmpDir().GetAbsolutePath()`
- LoD is only violated if the train of calls are performed on objects
 - then the ctx should provide a wrapper method to provide the same result
 - explosion in the number of methods
 - or it should be examined, why the result is needed
 - for example, if a new temp file is to be created, ctx should provide that instead
- However, if the links in the chain are data structures, LoD does not apply

Entities and data transfer objects

- Classes with only data, no behavior
- Private fields with public properties/accessors
- Useful in various technologies where behavior is intentionally separated from the data:
 - ORM
 - MVC
 - data transfer through network
- Make sure not to create hybrids: data structure with some business logic or behavior

Design considerations

- Sometimes it is better to separate behavior from data (procedural)
 - when new behavior is frequently added but the data structure is stable
 - database + business logic
 - when behavior is attachable or optional
 - visitor, strategy
- Sometimes it is better to keep related data and behavior together (OO)
 - when internal data representation can change
 - when new functions are rarely added
- Both of them are perfectly OK
 - just make sure to chose the approach that is the best for the job at hand

Summary

Summary

- Bad code:
 - hard to maintain
- Clean code:
 - easy to read, easy to understand
- Meaningful names:
 - shorter names are generally better than longer ones, so long as they are clear
 - a long descriptive name is better than a short enigmatic name
 - a long descriptive name is better than a long descriptive comment
 - the length of a name should correspond to the size of its scope
- Functions:
 - functions should be small
 - functions should do one thing
 - functions should not have more than 3 parameters
 - command-query separation

Summary

- **Comments:**

- proper use of comments is to compensate for our *failure to express ourselves in code*
- don't write misleading, redundant and noise comments
- comments should be informative, should clarify the code, should express design intent
- document the public API, but don't comment the internal implementation

- **Exceptions:**

- define exception classes in terms of a caller's needs
- don't return null
- don't pass null

- **Objects and data structures:**

- procedural code:
 - hard to add new data structures because all the functions must change
- object-oriented code:
 - hard to add new functions because all the classes in the hierarchy must change

API Design Guidelines

Objektumorientált szoftvertervezés

Object-oriented software design

Dr. Balázs Simon

BME, IIT

Outline

- Characteristics of a good API
 - API design process
 - API design guidelines
-
- Sources:
 - Stefan Nilsson: Thoughts on effective API design
<https://yourbasic.org/algorithms/your-basic-api/>
 - Matt Gemmell: API Design,
<http://mattgummell.com/api-design/>
 - Jasmin Blanchette: The Little Manual of API Design
<http://people.mpi-inf.mpg.de/~jblanche/api-design.pdf>

Why is API design important to developers?

- If you program, you are an API designer
 - good software is modular
 - each module has an API
- Useful modules are reused
 - once it has users, it can't be changed at will
- Following the API design principles improves code quality

Characteristics of a Good API

Easy to learn and memorize

- Consistent naming conventions and patterns
 - same name for the same concept
 - different names for different concepts
 - throughout the API
- A minimal API is easy to memorize
 - there is little to remember
- A consistent API is easy to memorize
 - you can reapply what you learned in one part of the API in a different part

Easy to learn and memorize

- The semantics should be simple and clear
- Follow the principle of least surprise
 - If a necessary feature has a high astonishment factor, it may be necessary to redesign the feature
- Don't force boilerplate code on the user
 - a "hello world" sample shouldn't be more than a few lines
- Convenience methods
 - they contribute to a bloated API
 - but they are accepted and encouraged if they are clearly documented and fit nicely with the API

Leads to readable code

- Code is written once but is read over and over again
- Readable code is easier to document and to maintain
- It is less likely to contain bugs, since bugs are made more visible by the code's readability
- Readable code can be concise and verbose, it does not matter
- It is at the right abstraction level
 - it does not hide important things
 - doesn't force specifying irrelevant information

Hard to misuse

- A good API makes it easier to write correct code than incorrect code
- It does not force the user to call methods in a specific order
- Users do not have to be aware of implicit side effects

Easy to extend

- Libraries grow over time
 - new classes
 - new methods
 - new parameters
 - new enum values
- APIs should be designed with this in mind
- Also provide extensibility for users of the API

Complete

- Ideally, an API should be complete and let users do everything they want
- In practice, this is never true
 - there will always be users who need something the API does not provide
- Therefore, it should be possible for users to extend and customize existing API
 - e.g. subclassing, overriding

Well documented

- The public API of a library must be documented
 - the internal implementation doesn't need documentation, it can follow clean-code rules
- For each documented element:
 - What is it?
 - What does it do? (Not how it works!)
 - short sentence stating the purpose
 - a slightly longer explanation
 - and finally, all the details

API Design Process

1. Know the requirements

- Collect the requirements
 - sometimes its easy: implement a standard
 - sometimes its hard: the requirements are unclear
- Ask as many people as possible
 - the most important is the target audience, the potential users
 - colleagues, boss, etc.
- Sometimes you will receive solutions instead of requirements
 - be careful, better solutions may exist

2. Write use cases before you write any other code

- The wrong order of design is:
 - implement the functionality -> design API
- An API like this usually reflects the structure of the underlying code rather than meeting the users' needs
- The implementation should adapt the user not the other way around
- Before you start implementing the API write code snippets how you would like to use it
 - don't worry at this stage that it will be hard to implement
 - they will be good for samples and unit tests, too

3. Look for similar APIs in the same library

- Look if you can find a similar API
 - e.g. if you would like an XmlQuery, you can look at SqlQuery
 - the other library has probably evolved by a lot of feedback
- Mimic the similar API in your API
 - users will be comfortable with it already
 - you can adopt years of design decisions
- Of course, if the other API is bad, don't follow it
 - but not every API is that bad, there can be useful ideas
 - also if applications have to be ported from the old API, consider being backwards compatible and implement a superset of the old API

4. Define the API before you implement it

- Specify the API and its semantics before you implement it
- It is better for users if the API is straightforward and the implementation is tricky than the other way around
- As you implement the API or write unit tests you might find flaws or undefined parts in your API design
 - correct these in the design
 - but never let implementation details leak into the API
 - on-disk and on-the-wire formats, exceptions, tuning parameters
 - except for optimization options, but be careful with these

5. Have your peers review your API

- Ask feedback
 - potential users
 - colleges, boss
- Momentarily forget that it will be hard to implement
- Negative feedback is also useful
- Every opinion should be considered
- Keep the specification of the API short
 - 1 page is ideal
 - it is easier for people to read
 - it is easier for you to maintain

6. Write several examples against the API

- Write a few examples that use the API
- Start it early
 - even before you implement it
 - even before the specification
- Use the use-cases defined earlier
- It will also help if other people write examples and give you their feedback
- These examples will be good for documentation, samples and unit tests
- Keep writing examples as the API evolves
- Eat your own dog food
 - use your own API over an extended period of time on different types of tasks and projects to know if it really works as intended
 - meanwhile see if you can make improvements, simplify things

7. Prepare for extensions

- The API will be extended in two ways:
 - by the maintainers of the API
 - by the users of the API
 - through inheritance and polymorphism
- Service Provider Interface (SPI)
 - plug-in interface enabling multiple implementations
- Planning for extensibility requires going through realistic scenarios
 - for each class that can be overridden write at least 3 different subclasses (as examples or as public classes of the API) to ensure that it is powerful enough to support a wide range of requirements

8. Don't publish internal APIs without review

- Sometimes APIs start as internal APIs
- Later they are published
- Make sure they are reviewed before publishing
 - e.g. misspellings

9. When in doubt, leave it out

- If you have doubts about a functionality
 - leave it out
 - or mark it as internal
 - you won't be able to please everyone
 - aim to displease everyone equally
- Wait for feedback from users
 - wait at least three independent users request something until you implement a new feature
- You can always add, but you can never remove
- Just say no
 - an API shouldn't encourage bad design decisions
- Expect to make mistakes
 - a few years of real-world use will flush them out
 - expect to evolve API

10. Don't change it

- A software library needs to be backwards compatible
- It is OK to:
 - improve documentation
 - change the implementation
 - introduce new features
- But don't change the API!
 - You will break other people's code!
 - You need to get your API right at the very first attempt!
- Use semantic versioning: major.minor.patch
 - increments:
 - major: when an incompatible API change is made
 - minor: when a functionality is added in a backwards-compatible manner
 - patch: when backwards-compatible bug fixes are made

API Design Guidelines

G1. Choose self-explanatory names and signatures

- Pick names and signatures that are self-explanatory
- The names should read like prose
- The meaning of arguments should be clear at the call
 - beware of bool values
 - beware of lots of values of the same type after each other
 - consider using enums
 - consider using parameter objects
- Be consistent with the order of parameters
- Use the terminology of your audience
- Use meaningful names for parameters

G2. Choose unambiguous names for related things

- Use consistent names
- Don't use different names for the same thing
 - e.g. Control-Widget, send-post
- If similar concepts have to be distinguished, use different names for them

G3. Beware of false consistency

- If you use a convention for something, then don't use this convention for something else
- Example:
 - prefix 'set' is for setters
 - don't use this convention for methods other than setters

G4. Avoid abbreviations

- Abbreviations are problematic because they have to be remembered what they mean and in what context
 - e.g. **GetWin** instead of **GetWindow**
- Exception: conventional abbreviations if they are obvious
 - **min**, **max**, **dir**, **prev**
 - just be consistent, and use the abbreviation everywhere
- Acronyms are OK, acronyms are not abbreviations
 - so use **XML** and not eXtensible Markup Language
 - also: **HTML**, **UML**, **IO**, etc.

G5. Prefer specific names to general names

- Prefer specific names even if they seem to be too restrictive
- Once a name is taken, it cannot be used for something else
- Later if you would like to generalize you can pick general names
- Example: `GetLength` is better than `GetInt`

G6. Use the local dialect

- APIs belong to a platform and a developer ecosystem
- Learn your target platform's conventions before coding
- Learn conventions for constructors, destructors, exceptions, method names, property names, memory management
- Use that terminology throughout your code
- Similarly, if you port an API to a new programming language, follow that language's conventions
 - the users of that language will be happy
 - the old users of the API won't mind

G7. Don't be a slave of an underlying API's naming practices

- If you need to wrap an existing API, don't hesitate to invent your own terminology
- The wrapped API may have poorly chosen names, or it clashes with the names of your library

G8. Choose good defaults

- Choose good defaults so that the users won't have to copy and paste boilerplate code to get started
- Good defaults also make the API simple and predictable
- Name boolean options so that they are default to false
 - e.g. **visible** vs. **hidden**
- Document the defaults

G9. Avoid making your APIs overly clever

- Don't have a too clever API
- Avoid unexpected side effects
 - e.g. setText() automatically recognizes HTML text and also calls setTextFormat()

G10. Consider performance consequences of API design decisions

- Bad decisions can limit performance
 - making a type mutable requires defensive copies
 - providing constructor instead of static factory
 - using implementation type instead of interface
- Do not warp API to gain performance
 - underlying performance issue will get fixed, but headaches will be with you forever
 - good design usually coincides with good performance

G11. Pay attention to edge cases

- Edge cases are important, since other cases build on them
- Usually edge cases don't have to be handled separately, they are usually the neutral element of the operation
 - e.g. 1 for multiplication, 0 for addition, empty collection, etc.
- If you write extra code for edge cases, be suspicious
 - e.g. returning null instead of empty collection
- Write unit tests for the edge cases

G12. Minimize mutability

- Classes should be immutable unless there's a good reason to do otherwise
 - advantages: simple, thread-safe, reusable
 - disadvantage: separate object for each value
- If mutable, keep state-space small, well-defined
 - make clear when it's legal to call which method
- Examples:
 - C#: `DateTime`
 - Java: `Date`

G13. Design and document for inheritance or else prohibit it

- Inheritance violates encapsulation
 - Subclass sensitive to implementation details of superclass
- If you allow subclassing, document self-use
 - How do methods use one another?
- Conservative policy: all concrete classes final/sealed

G14. Be careful when defining virtual APIs

- “Fragile base class problem”
- Difficult to get virtual base classes right
- Easy to break them between releases
- Mistakes:
 - too few virtual methods: cannot be reused
 - all methods are virtual: some methods are dangerous to reimplement
- Rules:
 - public methods should not be virtual
 - virtual methods should be protected
 - use template methods and don’t make them virtual

G15. Strive for property-based APIs for GUI

- Don't make users to initialize everything in constructors with long parameter list
 - only the required settings should be constructor parameters
- Use setters instead
 - choose appropriate default values so that the user only has to explicitly change what they need
 - the user doesn't need to remember the order of the values
 - the code is more readable, since it is visible, which property has which value
 - the values of the properties can be retrieved by getters
 - use lazy initialization because the properties can be set in any order

G16. Anticipate customization scenarios

- User interfaces should have sensible defaults to fit the majority
 - don't give the user options
- However, APIs should be customizable to be flexible
 - if you publish properties for customization, also publish related properties
 - e.g. ForegroundColor, BackgroundColor
- Decide what options will serve 70% or so of the usage situations you can think of, and provide those options
- Better to have a simple API with few options than a complicated API with a lot of options

G17. Avoid long parameter lists

- Three or fewer parameters is ideal
 - more and users will have to refer to docs
- Long lists of identically typed parameters harmful
 - programmers transpose parameters by mistake
 - programs still compile, run, but misbehave
- Two techniques for shortening parameter lists
 - break up method
 - create parameter object to hold parameters

G18. Use convenience methods

- Implementing the API as a developer you may add convenience methods
- They are usually kept hidden from the public interface of the API
- Consider publishing these convenience methods
 - if they are convenient for you, they may prove to be convenient for others, too
- Example:
 - Opening a file for reading:
 - C#: `reader = new StreamReader("file.txt")`
 - Java: `reader = new BufferedReader(new FileReader("file.txt"))`
 - Reading the contents of a file:
 - C#: `reader.ReadToEnd()`
 - Java: `new String(Files.readAllBytes(Path.get("file.txt")))`

G19. Get up and running in 3 lines

- The API should be able to used in 3 lines:
 - instantiation
 - basic configuration, setting properties
 - execution
- Anything more and the API has too much boilerplate code
 - boilerplate code spreads with copying (e.g. StackOverflow)

G20. Magic is OK. Numbers aren't.

- It is possible that you need special values to implement the API
- It is also possible that you have to apply tricks and magic in the implementation to keep the API nice for the users
- But never publish magic with the API
- Wrap special values in meaningful named constants, enumerations, etc.

G21. Fail fast – report errors as soon as possible after they occur

- Compile time is best - static typing, generics
- At runtime, first bad method invocation is best
 - check pre-conditions at the beginning of the methods
 - e.g. null checks, valid value ranges
 - methods should be failure-atomic
- Throw exceptions to indicate exceptional conditions
 - don't force client to use exceptions for control flow
 - conversely, don't fail silently

G22. Favor unchecked exceptions

- Checked: client must take recovery action
- Unchecked: programming error
- Overuse of checked exceptions causes boilerplate
- Checked exceptions violate OCP
- Include failure-capture information in exceptions
 - allows diagnosis and repair or recovery
 - for unchecked exceptions: message suffices
 - for checked exceptions: provide accessors

G23. Include resolution in the exception message

- Avoid exceptions with no messages
- Avoid leaking null-pointer exceptions
- Avoid swallowing exceptions
- Describe the problem in the exception message
- Provide meaningful description of the problem
- Provide resolution message: how to fix the problem
- Don't include sensitive information in exceptions
 - exceptions are usually logged
- Example:
 - Unhandled Exception: System.InvalidOperationException: Service 'Services.Company' has zero application (non-infrastructure) endpoints. This might be because no configuration file was found for your application, or because no service element matching the service name could be found in the configuration file, or because no endpoints were defined in the service element.

G24. Overload with care

- Avoid ambiguous overloads
 - multiple overloads applicable to same actuals
 - conservative: no two with same number of args
- Just because you can doesn't mean you should
 - often better to use a different name
- If you must provide ambiguous overloads, ensure same behavior for same arguments

G25. Test the hell out of it

- Test your API
- Use the use-cases and samples written earlier
- Write unit tests and regression tests
- Write tests for the edge cases
- Don't publish a buggy API
 - users won't trust a buggy API
 - they may soon abandon it

G26. Document your API

- An API without documentation is painful to use
 - e.g. a lot of open source libraries
- Documentation in comments is a useful way to do this
 - it should be sufficiently detailed but not too verbose
 - class: what an instance represents
 - method: contract between method and its client
 - pre-conditions, post-conditions, side-effects
 - parameter: indicate units, form, ownership
 - document default values
 - document state space
- Provide HTML and PDF documentation
 - HTML is easier to browse
 - PDF can be easier to read in order

Summary

Summary

- Characteristics of a good API
 - API design process
 - API design guidelines
-
- Sources:
 - Stefan Nilsson: Thoughts on effective API design
<https://yourbasic.org/algorithms/your-basic-api/>
 - Matt Gemmell: API Design,
<http://mattgummell.com/api-design/>
 - Jasmin Blanchette: The Little Manual of API Design
<http://people.mpi-inf.mpg.de/~jblanche/api-design.pdf>

Distributed OO

Objektumorientált szoftvertervezés

Object-oriented software design

Dr. Balázs Simon

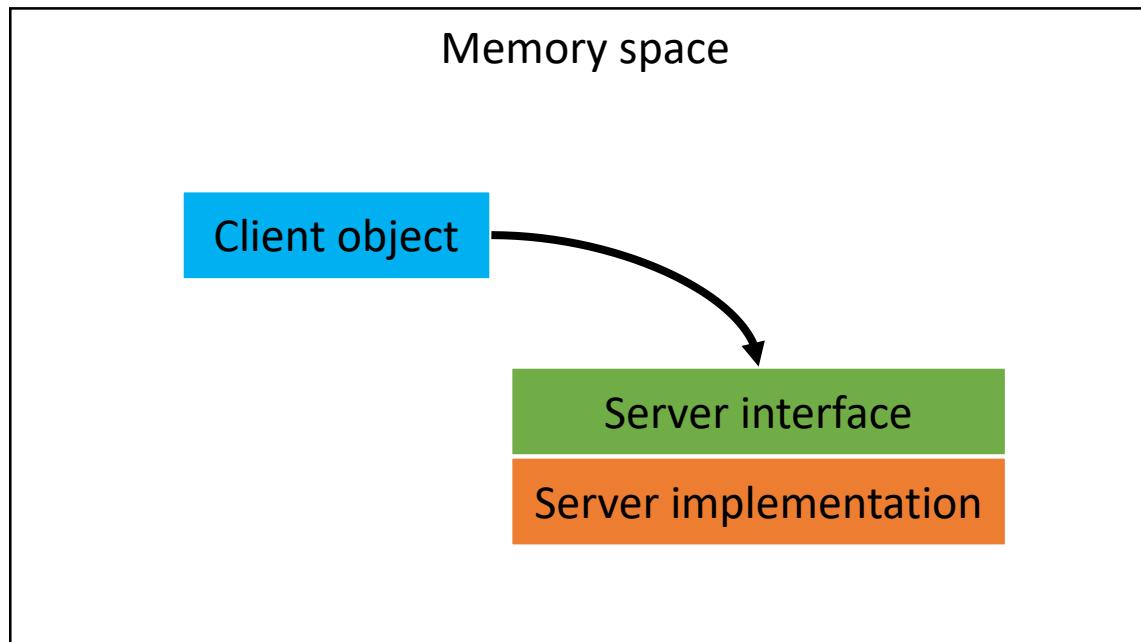
BME, IIT

Outline

- Remote communication
- Problems introduced by remote communication
- Possible solutions
- Technologies for remote communication
 - SOAP, REST

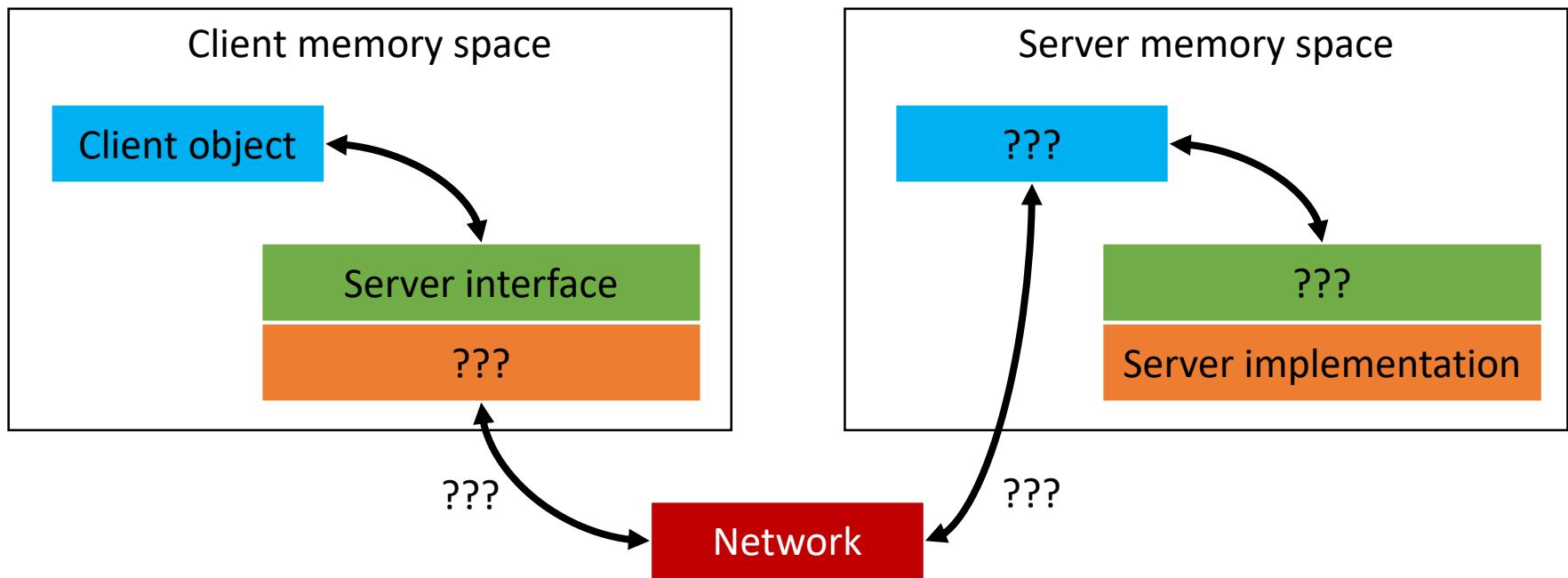
Local call

- Caller object: client
- Called object: server
- They are in the same memory space:



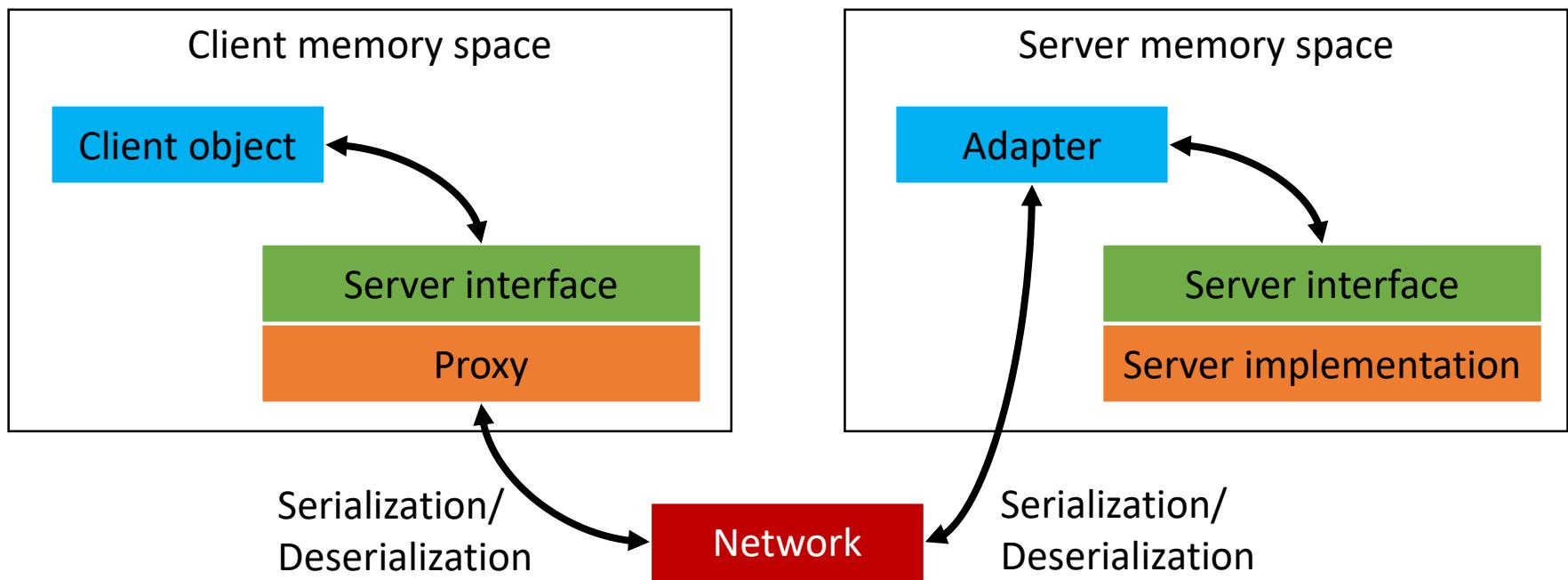
Remote call

- Caller object: client
- Called object: server
- They are in different memory spaces:



Remote call

- Proxy (stub): appears as if the server was local
 - connects to server, serializes parameters, deserializes result
- Adapter: publishes the server implementation on the network
 - accepts client requests, deserializes parameters, calls server implementation, serializes result



Problems introduced by remote communication

- Memory management problems
 - separate memory spaces for the client and the server
 - parameters and results have to be serialized and deserialized
 - pointers/references have to be serialized recursively
 - memory allocation and freeing
 - preserving server state between calls
- Network problems
 - problem if the server is unavailable
 - problem if the client is unavailable
 - data integrity
- Concurrency problems
 - multiple clients
- Latency problems
 - large response times
 - long running operations
 - synchronous and asynchronous calls

Questions to answer

- How do we define the interface of the server?
- How do we find the server?
- How do we implement the proxy?
- How do we implement the adapter?
- How do we serialize and deserialize data?
- How do we manage memory?
- How can we serve multiple clients?
- How many server object instances do we need?
- How do we preserve state between calls?
- How can we communicate if the server or the client is unavailable?
- How do we handle synchronous calls?
- How do we handle asynchronous calls?

Although these questions arise from distributed communication, the answers can also be used in a single application to decrease coupling between components/classes, especially when implementing a server application.

How do we define the interface of the server?

- 1. The client and the server are from the same language:
 - use the interface construct of the language
- 2. The client and the server are from different languages:
 - we need a language independent interface descriptor
 - generate language specific interfaces from this descriptor

How do we find the server?

- 1. The client knows where the server is
 - address burned into or configured for the client
- 2. The server has a logical name, which is registered into a naming service (white pages)
 - client has to find the naming service first
 - then it asks for the physical address of the server by its logical name
- 3. The server implements an interface, which is registered into a trading service (yellow pages)
 - client has to find the trading service first
 - then it asks for the physical addresses of servers which implement a the required interface

How do we implement the proxy?

- Proxy + adapter design pattern
 - usually (but not necessarily) lazy initialization
 - converts high level method calls to network bytes
- Tasks of the proxy:
 - find the server
 - connect to the server
 - serialize parameters
 - deserialize result
- How do we implement the proxy?
 - we don't, it is usually provided by a framework
 - it can be generated off-line or at run-time from the interface descriptor

How do we implement the adapter?

- Adapter design pattern
 - converts network bytes to high level typed method calls
- Tasks of the adapter:
 - accept client connections
 - serve client requests
 - instantiate server object
 - dispatch requests to method calls
 - handle multiple parallel requests
 - deserialize parameters
 - serialize result
- How do we implement the adapter?
 - we don't, it is usually provided by a framework
 - it can be generated off-line or at run-time from the server interface + server implementation

How do we serialize and deserialize data?

- Serialization is usually not done by hand, it is provided automatically by a framework
 - annotations may be needed to customize serialization
 - annotated types can be generated from a language independent interface descriptor
- Serialization:
 - always deep copy
 - pointers/references, in-out and out parameters behave differently than in local calls

How do we serialize and deserialize data?

- 1. Binary serialization
 - fast and efficient
 - uses less memory
 - requires binary compatibility between the client and the server
 - usually when the client and the server are from the same language
- 2. Text serialization
 - slow and not so efficient
 - may need a lot of memory
 - has much better compatibility than binary serialization
 - even between different languages

How do we manage memory?

- 1. In modern programming languages with GC we don't have to
 - memory is managed by the garbage collector
- 2. In older programming languages with no GC
 - usually very complex
 - who allocates?
 - who frees?
 - depends on the direction of the parameter
 - in, out, in-out or return
 - memory management may be part of the framework API
 - lower abstraction level leaks up
 - complexity may be mitigated by smart pointers

How can we serve multiple clients?

- 1. Single threaded server
 - no concurrency problems
 - A. Blocking
 - simple
 - clients may have to wait a lot
 - resource utilization is not very efficient
 - B. Non-blocking
 - a bit more complex programming model
 - clients do not have to wait
 - efficient resource utilization
- 2. Multi-threaded server
 - concurrency problems
 - A. Separate thread for each client
 - thread creation overhead
 - may overload the server
 - B. Thread pool
 - no thread creation overhead
 - a scheduler is needed
 - service operations should not occupy a thread for a long time

How many server object instances do we need?

- 1. One:
 - singleton, every client sees the same server object
 - no client-specific state
- 2. One for each client:
 - every client sees its own server object
 - we can have client-specific state
 - but:
 - When can we free the server object?
 - What if the server restarts?
- 3. Multiple but not bound to any specific client:
 - subsequent client requests may be served by different server objects
 - no client-specific state

How do we preserve state between calls?

- 1. Store state in server memory
 - we need a specific server object instance for each client
 - subsequent client requests must find this same server object
 - problem: not scalable
 - How do we find the same server object?
 - What if we run out of memory?
 - What if the server restarts?
- 2. Store state on the client side and pass it in every call
 - it does not matter how many server object instances exist
 - scalable
 - problem: the state may be large to transfer in every call
- 3. Store state in a database
 - it does not matter how many server object instances exist
 - scalable
 - the client may need to identify itself (e.g. send a session id)

How can we communicate if the server or the client is unavailable?

- 1. Client needs an immediate result
 - synchronous call
 - we don't communicate, since we can't wait
- 2. Client doesn't need an immediate result
 - asynchronous call
 - usually a long running operation
 - A. No third party
 - we don't communicate, since we can't
 - B. Reliable third party
 - client sends requests to a reliable third party
 - server reads requests from the third party, and sends the reply back through this third party
 - problem: we need to match requests with responses on the client side

How do we handle synchronous calls?

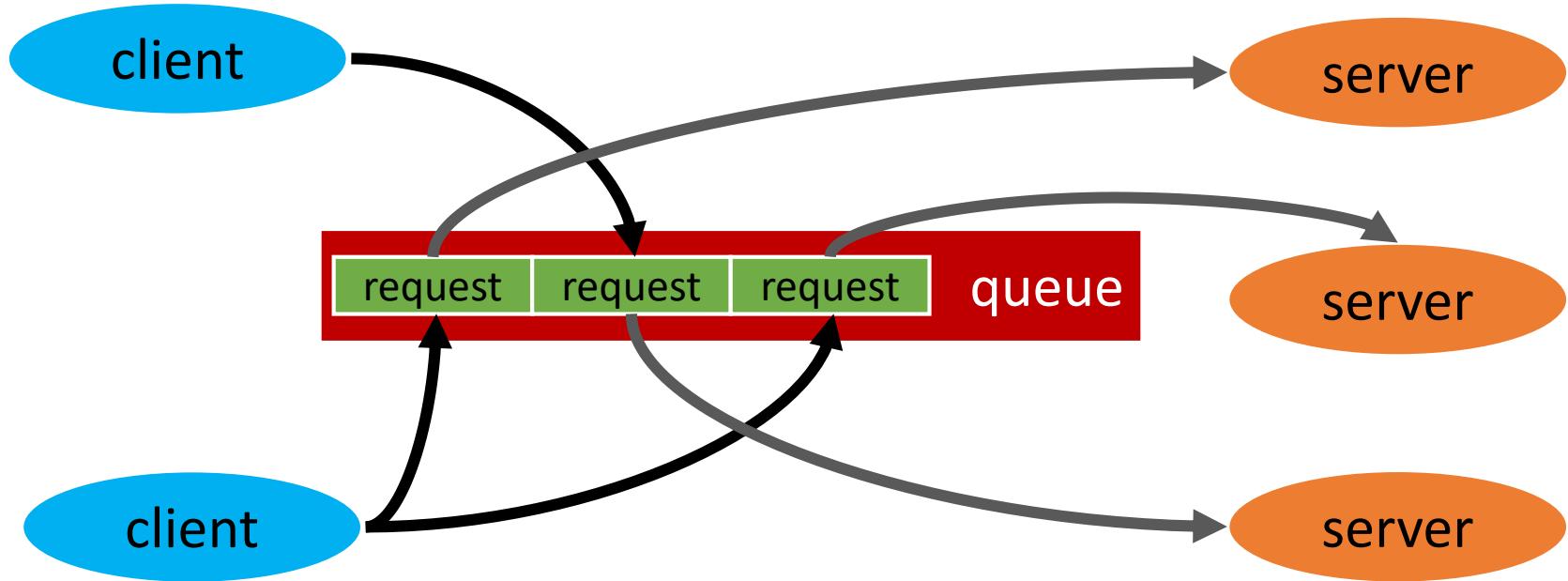
- Client sends request and waits for the result (blocks until the result arrives)
- Server accepts request and serves it:
 - threads:
 - single thread
 - separate thread for each request
 - thread pool
 - handling:
 - synchronous: serving is done on the accepting thread
 - asynchronous: serving is done asynchronously in the background

How do we handle asynchronous calls?

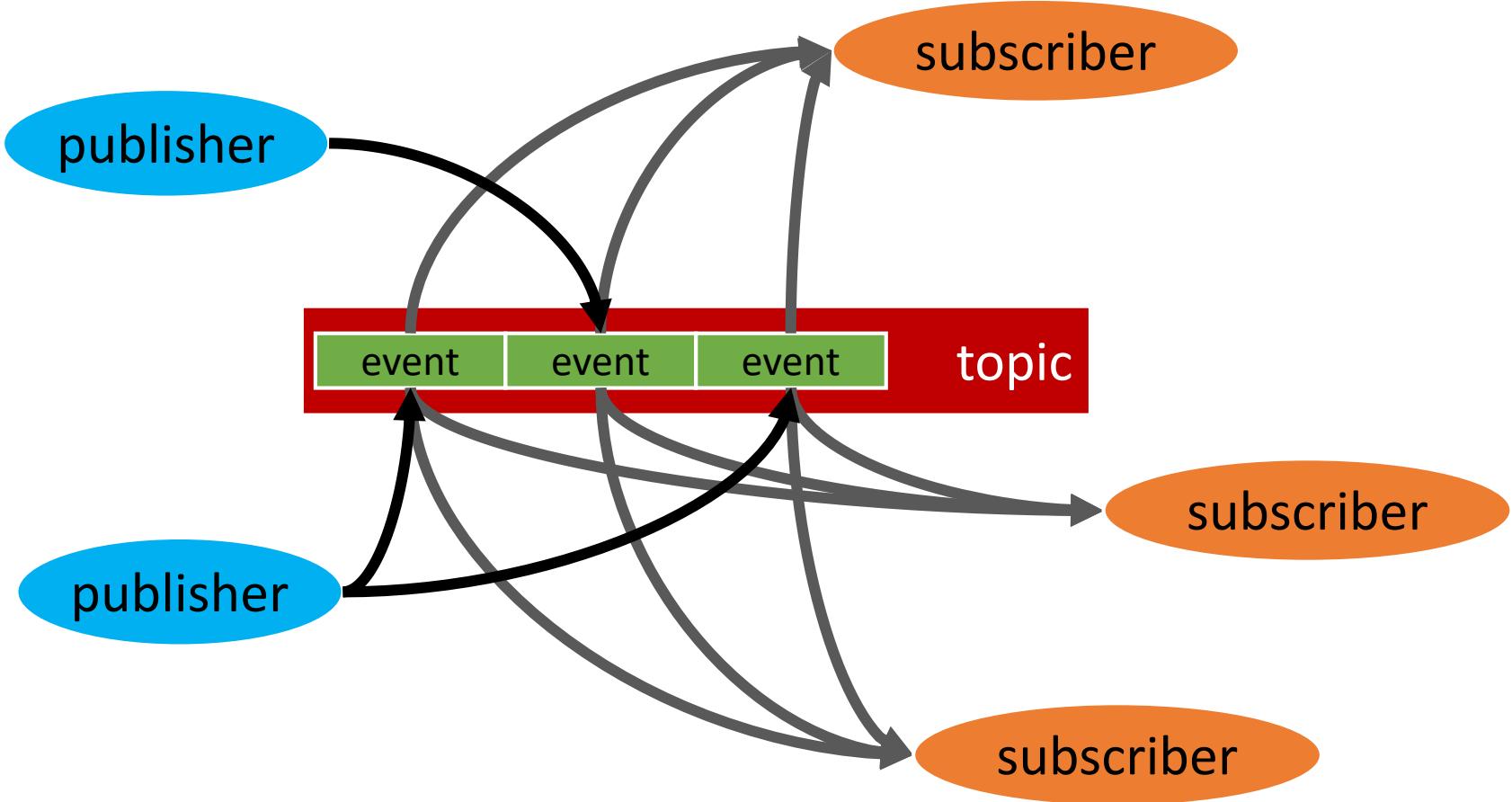
■ Asynchronous patterns:

- requests: usually many sources (clients) and one target (server)
 - non-blocking method calls
 - usually the requests are stored in a queue
 - if there are multiple instances of the server, each request is processed by exactly one instance
 - somehow the client needs to retrieve the result
 - 1. waiting: client blocks until the server is ready
 - 2. polling: client polls the server
 - 3. callback: client registers a callback method, server notifies the client
- events: usually one source (publisher) and many targets (subscribers)
 - publish-subscribe: targets subscribe to events, all of them are notified about a new event published by the source
 - events are stored in a queue (usually called topic or channel)
 - if there are multiple subscribers, each event is processed by all of them
 - filtering is also possible
 - usually there is no result, the publisher does not care about the event after it is sent

Asynchronous requests: client-server



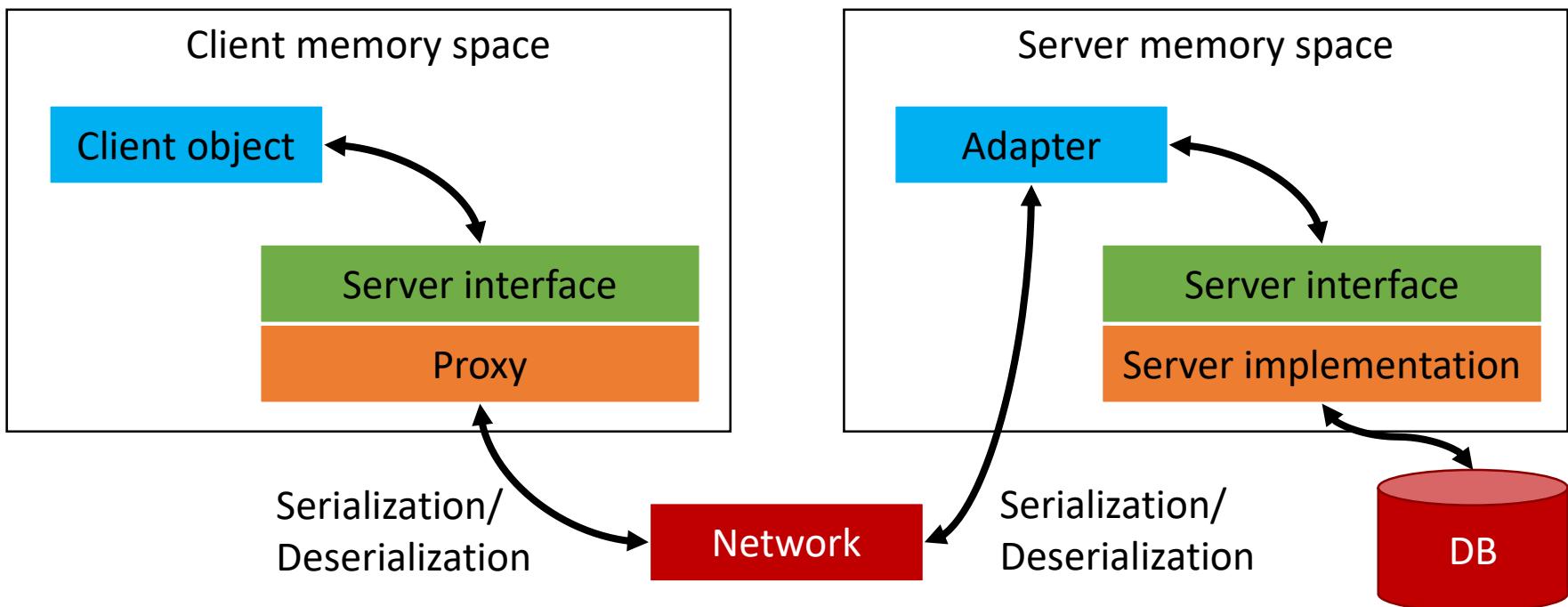
Asynchronous events: publish-subscribe



Technologies for remote communication

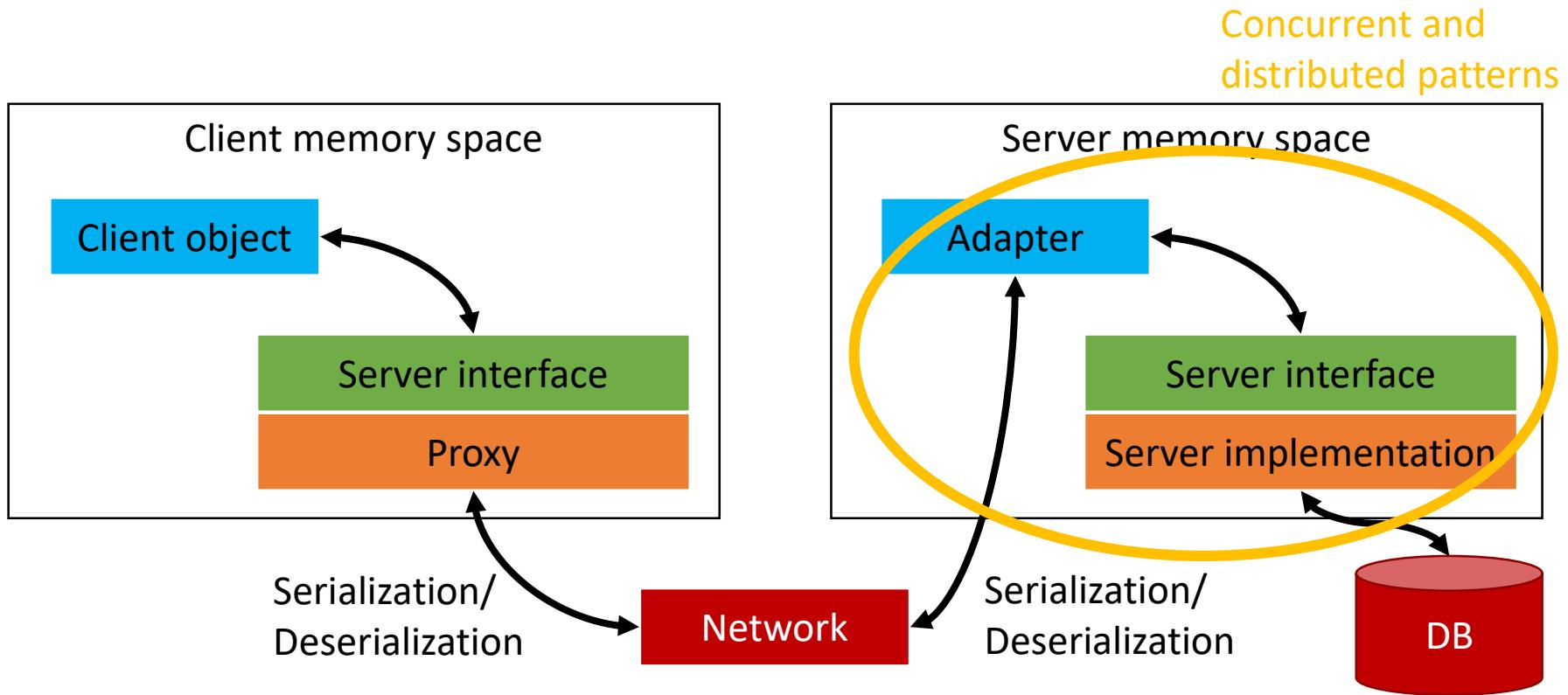
Remote call

- Caller object: client
- Called object: server
- They are in different memory spaces:



Remote call

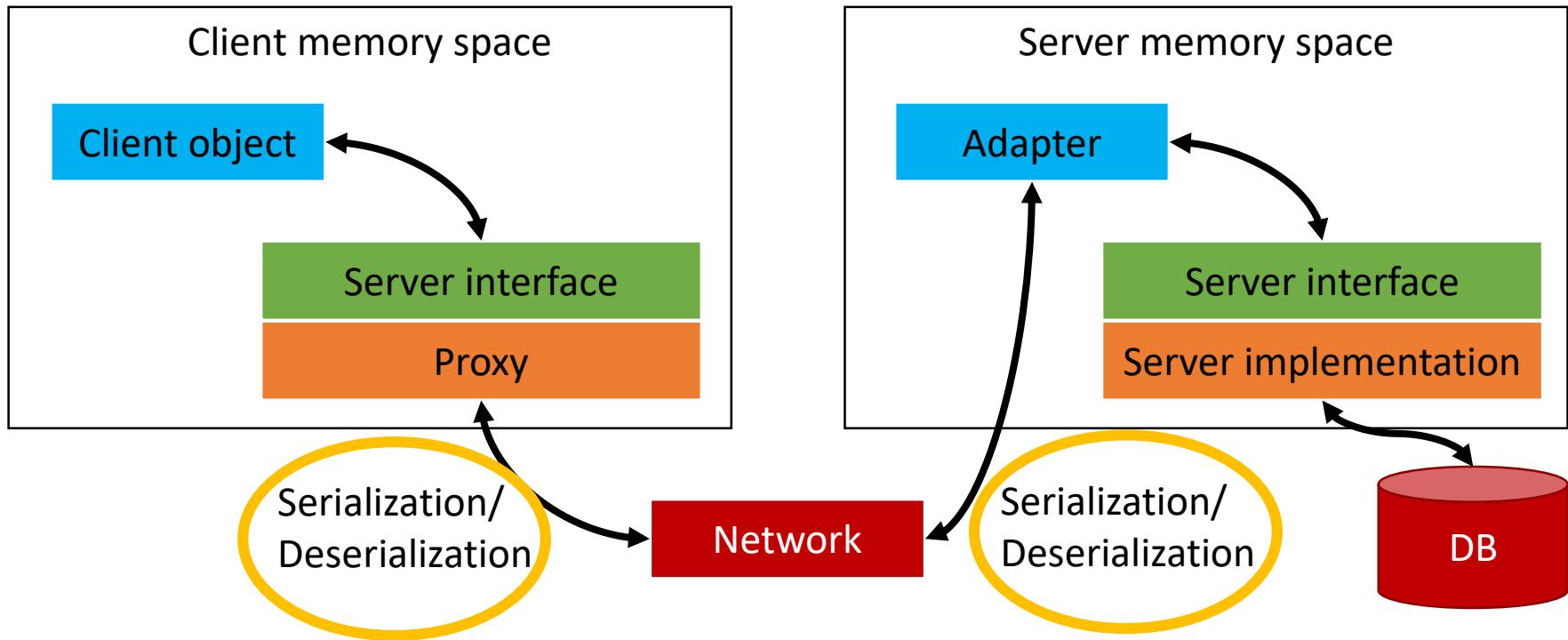
- Concurrent and distributed patterns:
 - implementing a multi-threaded server



Remote call

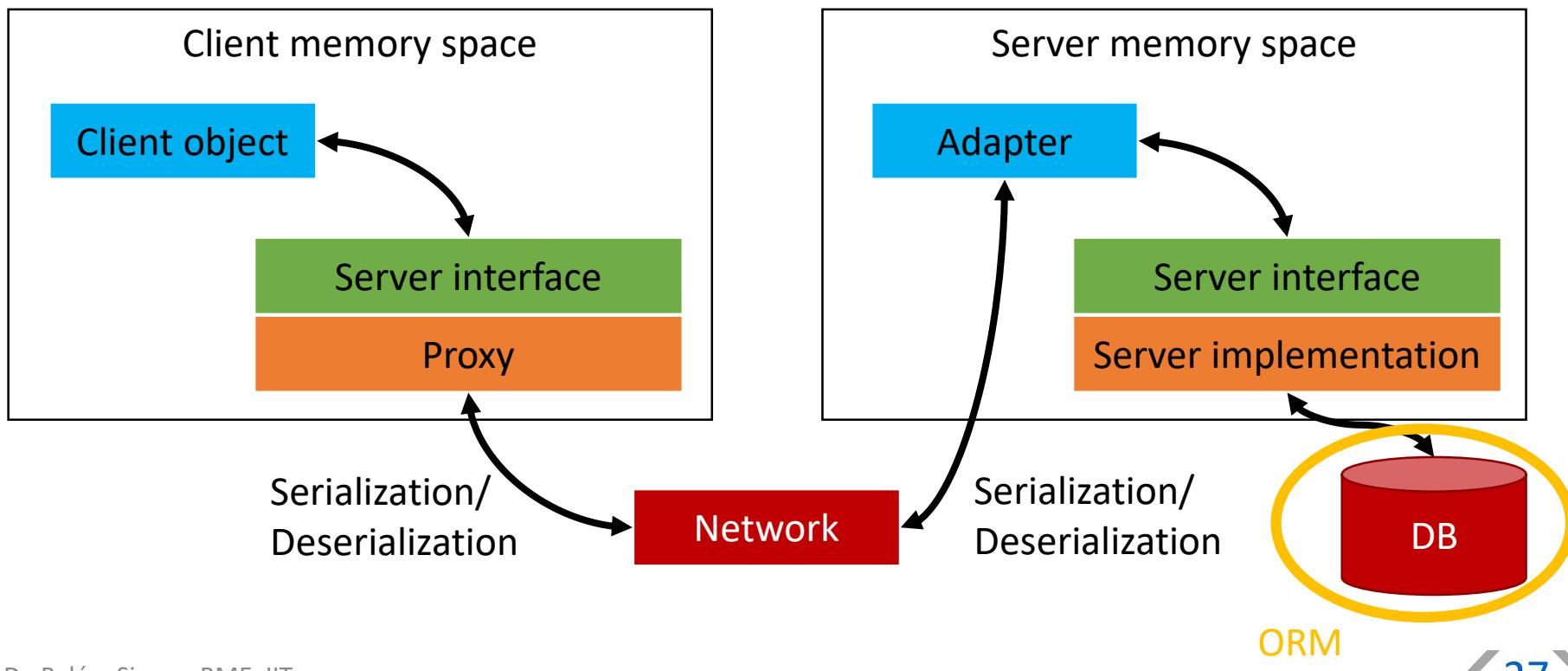
■ Serialization:

- binary: Java / .NET serialization
- text: XML, JSON
 - Java: Java API for XML-Binding (JAXB)
 - .NET: Windows Communication Foundation (WCF) DataContract



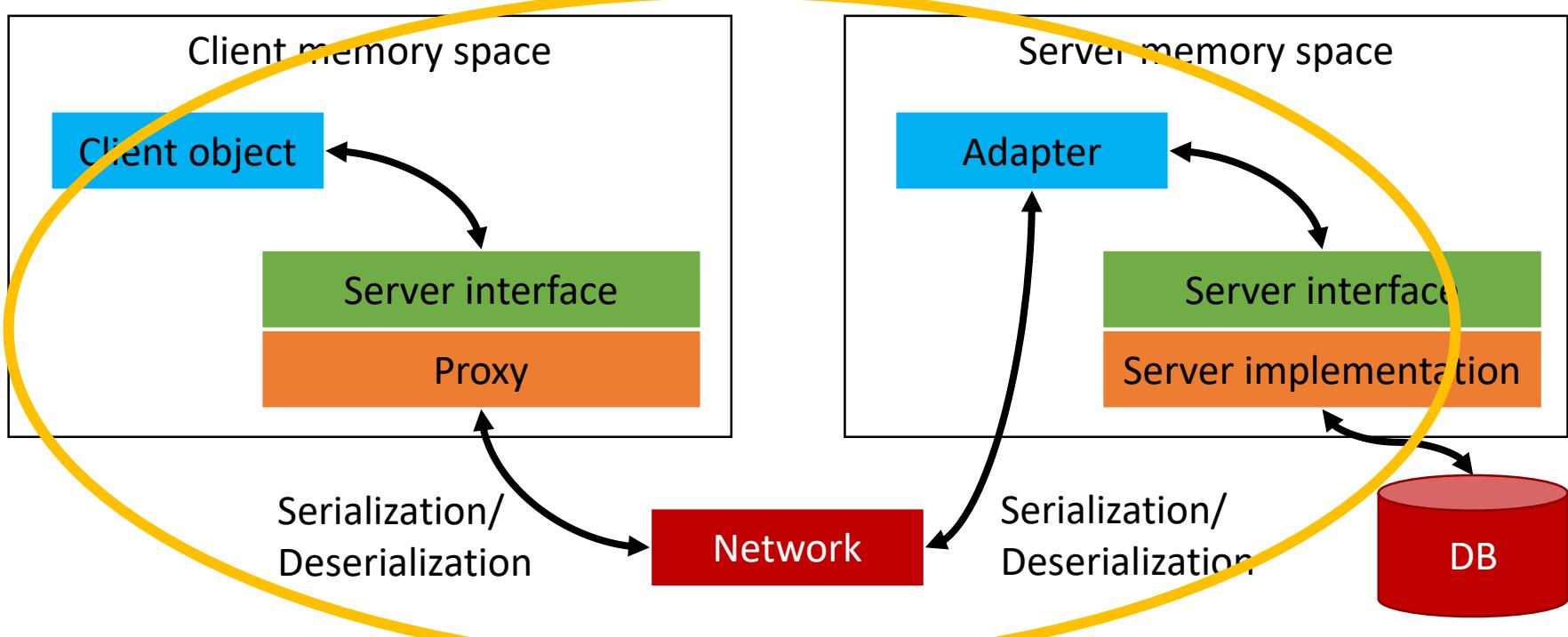
Remote call

- Object-relational mapping:
 - objects saved into a database (also a kind of serialization)
 - technologies:
 - Java Persistence API (JPA)
 - .NET EntityFramework



Remote call

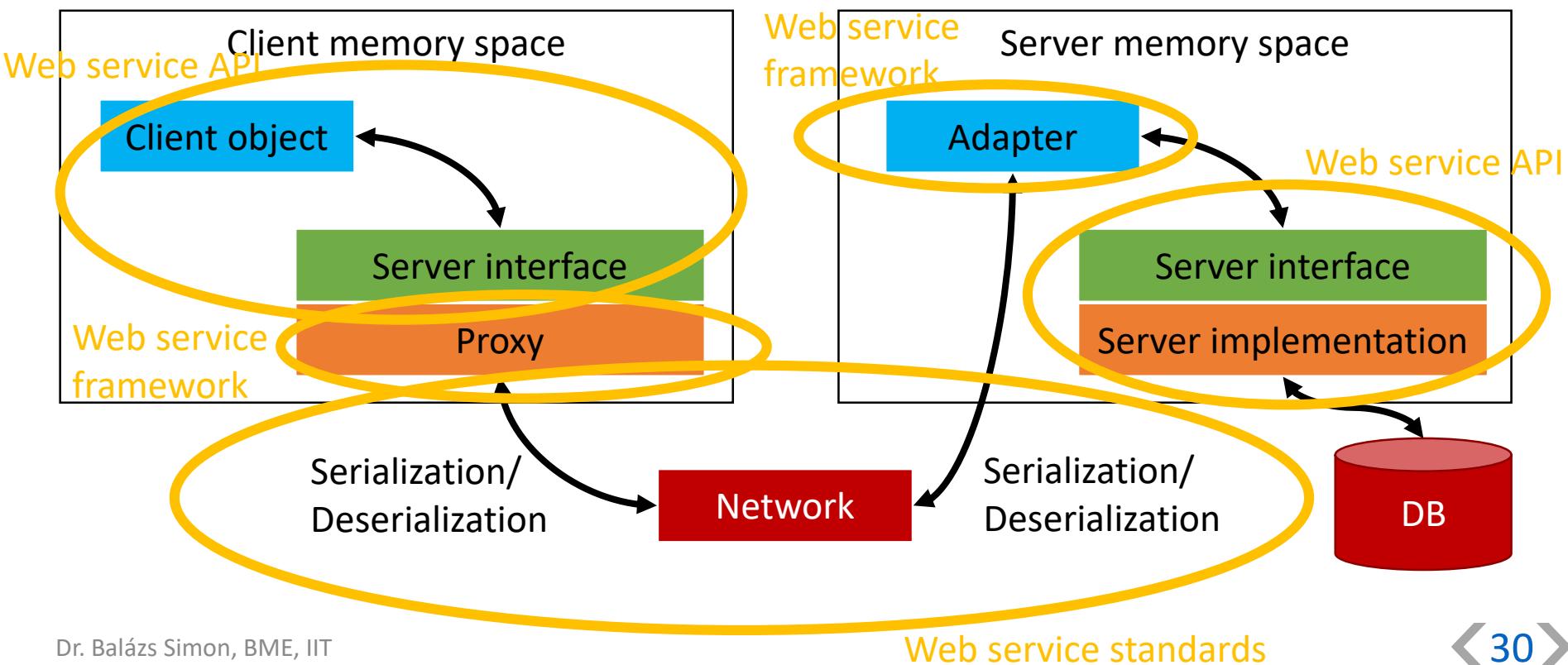
- Remote communication technologies:
 - Remote Method Invocation (RMI): binary, only within the same framework (Java / .NET)
 - SOAP web services: XML, even between different frameworks and languages
 - Java: Java API for XML-based web services (JAX-WS)
 - .NET: Windows Communication Foundation (WCF)
 - REST services: XML, JSON, etc., even between different frameworks and languages
 - Java: Java API for RESTful Web Services (JAX-RS)
 - .NET: Windows Communication Foundation (WCF)



SOAP web services

SOAP web services

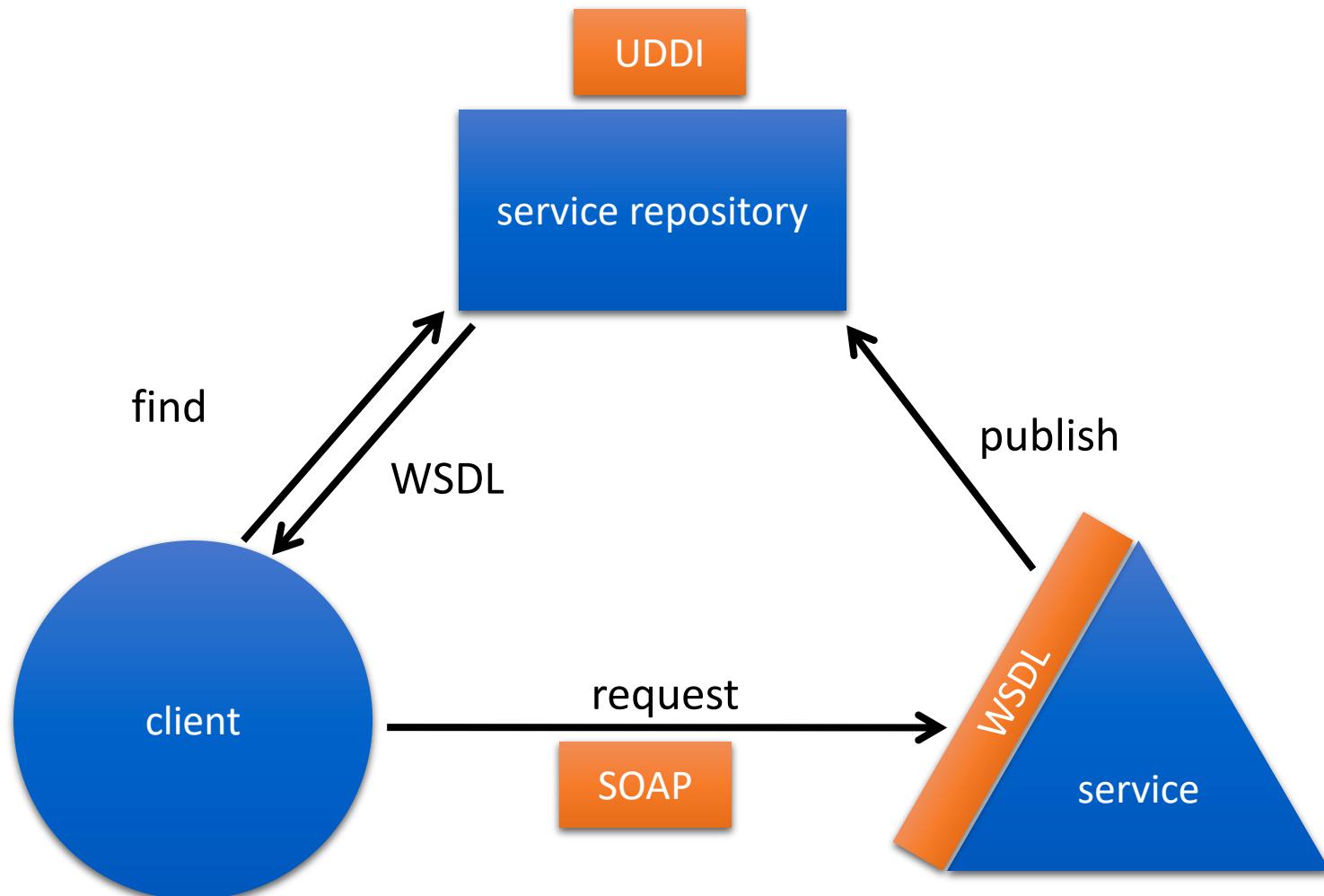
- XML between different frameworks and languages
 - Java: Java API for XML-based web services (JAX-WS)
 - .NET: Windows Communication Foundation (WCF)



SOAP web services

- A technology for creating services
- Standardized (OASIS, W3C)
- Independent of programming languages, frameworks and operating systems
- XML-based
- Transport protocol: usually HTTP
- Widespread, widely supported
- Various standards for middleware tasks:
 - addressing, reliable messaging, security, transactions

SOAP web services



WSDL = Web-Services Description Language

SOAP = Simple Object Access Protocol

UDDI = Universal Description, Discovery and Integration

Web services

- Definition:
 - service available through SOAP messages
- Message format:
 - SOAP = Simple Object Access Protocol
 - Versions: 1.1 and 1.2
- Interface descriptor:
 - WSDL = Web-Services Description Language
 - Versions: 1.1 and 2.0
- Service repository:
 - UDDI = Universal Description Discovery and Integration
 - Versions: 2.0 and 3.0

Example: SOAP request

```
interface IHelloWorld
{
    string SayHello(string name);
}
```

```
<s:Envelope
    xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
<s:Body>
    <SayHello xmlns="http://www.iit.bme.hu/soi">
        <name>me</name>
    </SayHello>
</s:Body>
</s:Envelope>
```

Example: SOAP response

```
interface IHelloWorld
{
    string SayHello(string name);
}
```

```
<s:Envelope
    xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
<s:Body>
    <SayHelloResponse xmlns="http://www.iit.bme.hu/soi">
        <SayHelloResult>Hi: me</SayHelloResult>
    </SayHelloResponse>
</s:Body>
</s:Envelope>
```

Example: WSDL 1/7

```
interface IHelloWorld
{
    string SayHello(string name);
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<wsdl:definitions
    targetNamespace="http://www.iit.bme.hu/soi"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
    xmlns:tns="http://www.iit.bme.hu/soi"
    xmlns:ns0="http://www.iit.bme.hu/soi">
    ...
</wsdl:definitions>
```

Example: WSDL 2/7

```
interface IHelloWorld
{
    string SayHello(string name);
}
```

...

```
<wsdl:types>
```

```
<xs:schema targetNamespace="http://www.iit.bme.hu/soi"
    xmlns:tns="http://www.iit.bme.hu/soi"
    xmlns:ns0="http://www.iit.bme.hu/soi"
    elementFormDefault="qualified">
    <xs:element name="SayHello" nillable="true"
        type="ns0:SayHello"/>
    <xs:complexType name="SayHello">
        <xs:sequence>
            <xs:element name="name" type="xs:string"
                nillable="true"/>
        </xs:sequence>
    </xs:complexType>
```

Example: WSDL 3/7

```
interface IHelloWorld
{
    string SayHello(string name);
}
```

...

```
<xs:element name="SayHelloResponse" nillable="true"
            type="ns0:SayHelloResponse"/>
<xs:complexType name="SayHelloResponse">
    <xs:sequence>
        <xs:element name="SayHelloResult" type="xs:string"
                    nillable="true"/>
    </xs:sequence>
</xs:complexType>
</xs:schema>
</wsdl:types>
```

• • •

Example: WSDL 4/7

```
interface IHelloWorld
{
    string SayHello(string name);
}

...
<wsdl:message name="IHelloWorld_SayHello_InputMessage">
    <wsdl:part name="parameters" element="ns0:SayHello"/>
</wsdl:message>

<wsdl:message name="IHelloWorld_SayHello_OutputMessage">
    <wsdl:part name="parameters",
                element="ns0:SayHelloResponse"/>
</wsdl:message>

...
```

Example: WSDL 5/7

```
interface IHelloWorld
{
    string SayHello(string name);
}
```

...

```
<wsdl:portType name="IHelloWorld">
    <wsdl:operation name="SayHello">
        <wsdl:input wsaw:action=
            "http://www.iit.bme.hu/soi/IHelloWorld/SayHello"
            message="ns0:IHelloWorld_SayHello_InputMessage"/>
        <wsdl:output wsaw:action=
            "http://www.iit.bme.hu/soi/IHelloWorld/SayHelloResponse"
            message="ns0:IHelloWorld_SayHello_OutputMessage"/>
    </wsdl:operation>
</wsdl:portType>
```

Example: WSDL 6/7

...

```
<wsdl:binding name="IHelloWorld_BasicHttpBinding_Binding"
               type="ns0:IHelloWorld">
  <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="SayHello">
    <soap:operation style="document" soapAction=
      "http://www.iit.bme.hu/soi/IHelloWorld/SayHello"/>
    <wsdl:input>
      <soap:body use="literal"/>
    </wsdl:input>
    <wsdl:output>
      <soap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
```

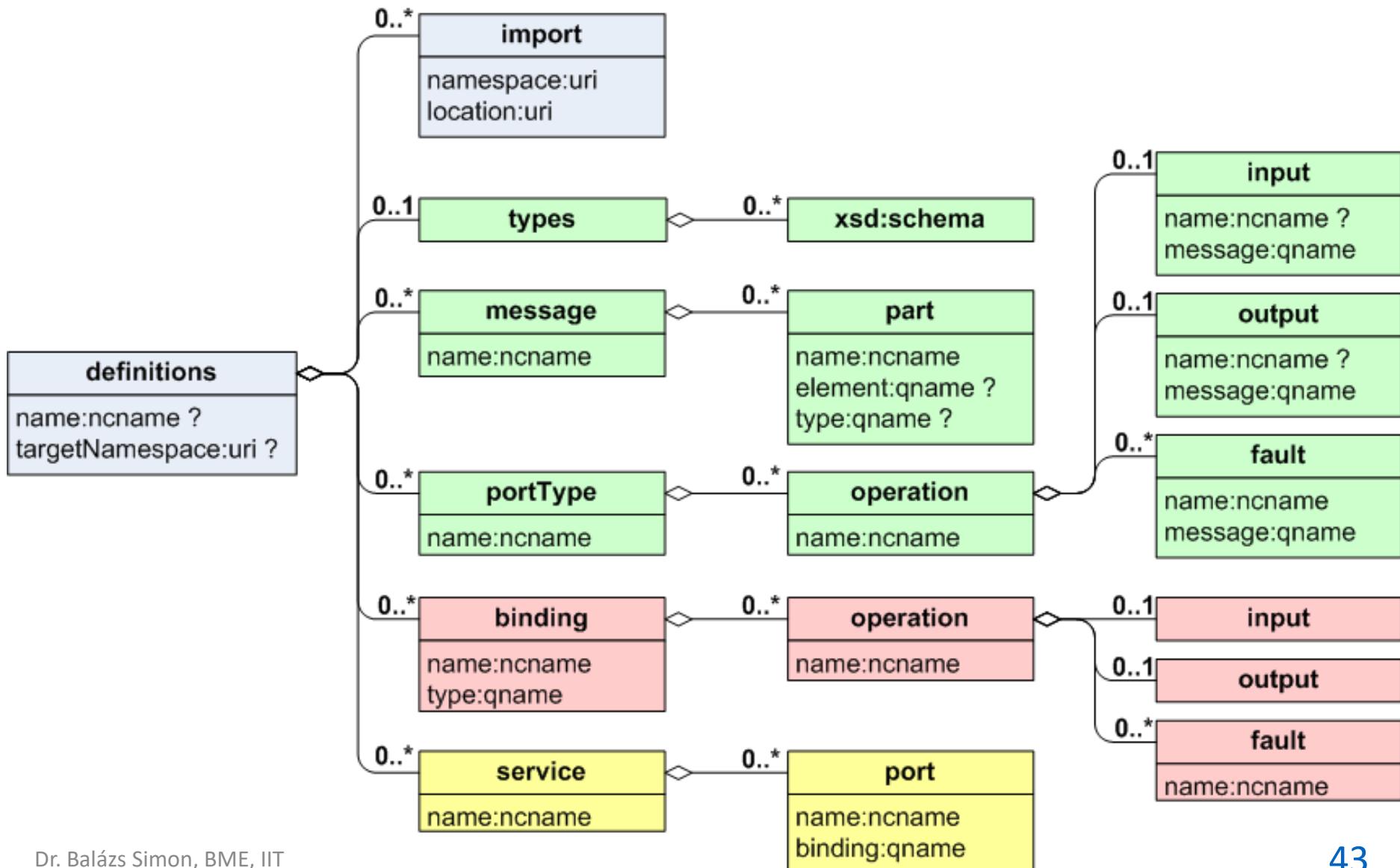
Example: WSDL 7/7

```
interface IHelloWorld
{
    string SayHello(string name);
}
```

...

```
<wsdl:service name="HelloWorld">
    <wsdl:port name="IHelloWorld_BasicHttpBinding_Port"
        binding="ns0:IHelloWorld_BasicHttpBinding_Binding">
        <soap:address location=
            "http://www.iit.bme.hu/Services/HelloWorld"/>
    </wsdl:port>
</wsdl:service>
```

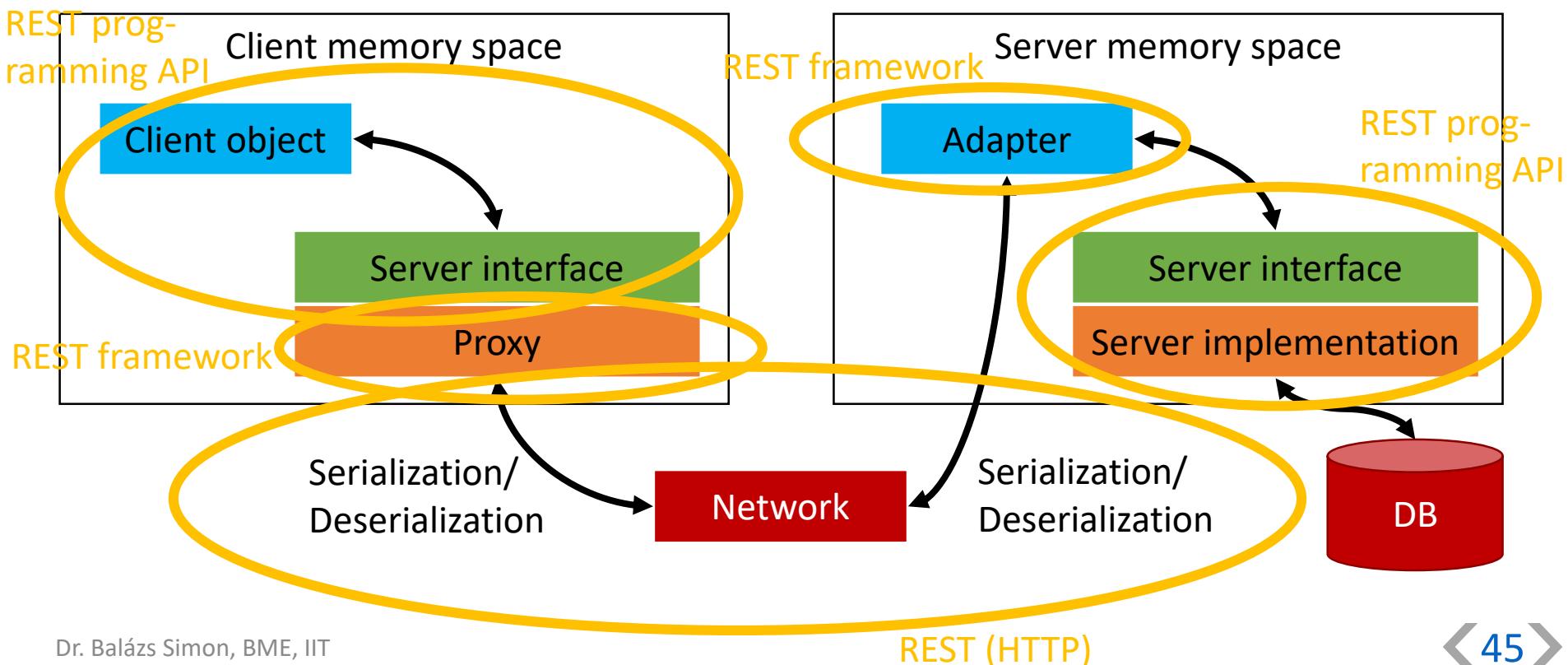
WSDL 1.1 summary



REST web services

REST services

- XML, JSON, etc., between different frameworks and languages
 - Java: Java API for RESTful Web Services (JAX-RS)
 - .NET: Windows Communication Foundation (WCF)



HTTP GET



Request:

GET /index.html HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0
Connection: keep-alive

Response:

HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Server: gws
Content-Length: 10200

HTTP status code

<!doctype html><html><head>...

HTTP GET: http://www.abc.com/login?user=xy&pass=123

HTTP Method

Local URL

Version

GET /login?user=xy&pass=123 HTTP/1.1

Host: www.abc.com

User-Agent: Mozilla/5.0

Connection: keep-alive

Headers+
Empty line

Host name

Query params

HTTP POST: http://www.abc.com/login

HTTP Method

Local URL

Version

POST /login HTTP/1.1

Host: www.abc.com

User-Agent: Mozilla/5.0

Content-Type: application/x-www-form-urlencoded

Content-Length: 16

user=xy&pass=123

Headers+
Empty line

Host name

HTTP Body:
Post parameters

REST

- REpresentational State Transfer
- RESTful HTTP
- HTTP protocol extension
 - GET, POST, PUT, DELETE
- Input parameters:
 - URL part
 - URL query string
 - POST parameter
 - HTTP body
- Result:
 - HTTP body
- Very simple: testable by browser

GET examples

- GET /api/movies
 - returns all movies
- GET /api/movies/12
 - returns the movie with identifier 12
- GET /api/movies/12/actors
 - returns the list of actors for the movie with identifier 12
- GET /api/movies?orderby=title
 - returns all movies sorted by their titles

GET

- GET is for reading, retrieving resources
- Must not modify any resources!
 - HTTP specification for GET
- Input parameters:
 - usually in the URL or in query parameter
 - identifiers, paging, filtering and sorting criteria
 - HTTP body is usually empty
- Output:
 - in the HTTP body, usually XML or JSON
 - success: 200 (OK)
 - error: 404 (Not found) or 400 (Bad request)

POST examples

■ POST /api/movies

- creates a new movie
- the content of the movie is passed in the HTTP body:

```
{  
    "title": "Batman Begins",  
    "year": 2005,  
    "director": "Christopher Nolan"  
}
```

POST

- POST is for creating new resources
 - usually for creating a new item under an existing resource item
- Care must be taken when resending POST requests
 - HTTP specification (e.g. credit card transaction)
 - the server creates the resource whenever a POST request is made
 - multiple identical POST requests may result in more than one resources with the same content
- Input:
 - URL: location of the parent resource
 - HTTP body: the content of the child resource to be created
- Output:
 - usually the identifier or location (Location HTTP header) of the resource created
 - success: 201 (Created)
 - error: 404 (Not found) or 409 (Conflict)

PUT examples

■ PUT /api/movies/12

- updates the movie with identifier 12
- or creates a new movie if it does not exist
- the content of the movie is passed in the HTTP body:

```
{  
    "title": "Batman Begins",  
    "year": 2005,  
    "director": "Christopher Nolan"  
}
```

PUT

- PUT is for updating a resource
 - or creating a new one if it did not exist
- Input:
 - URL of the resource to be updated
 - the URL contains the identifier of the resource
 - this will be the identifier on creation
 - so no multiple resources are created when repeating the PUT operation
 - HTTP body: the new content of the resource
- Output:
 - HTTP body may be empty
 - no identifier or location is necessary
 - success: 204 (No content), 201 (Created), 200 (OK)
 - error: 404 (Not found)

DELETE examples

- **DELETE /api/movies/12**
 - deletes the movie with identifier 12
- **DELETE /api/movies/12/actors/65**
 - deletes the actor 65 from the movie 12

DELETE

- DELETE is for deleting a resource
- Input:
 - URL of the resource to be deleted
 - the URL contains the identifier of the resource
 - HTTP body is usually empty
- Output:
 - either an empty HTTP body
 - or the content of the deleted resource
 - careful with this: it may be very large
 - success: 204 (No content) or 200 (OK)
 - error: 404 (Not found)

Input parameters

- Query parameter:
 - `http://.../calculator/add?left=3.0&right=5.0`
- Path parameter:
 - `http://.../calculator/add/3.0/5.0`
- Matrix parameter:
 - `http://.../calculator/add;left=3.0;right=5.0`
- POST parameter:
 - `http://.../calculator/add`
 - like the query parameter but inside the `HTTP body`
- HTTP body:
 - a serialized resource (e.g. in XML or JSON)
- Expected result data format: HTTP “Accept” header
 - for XML: `application/xml`
 - for JSON: `application/json`

Output result

- In HTTP body
- Actual data format: HTTP “Content-Type” header
 - or if the server could not support the requested data format:
HTTP 406 (Not acceptable)
- Usual data formats:
 - XML
 - JSON

Interface description

- No standard and widely supported interface descriptor
- Possible interface descriptors:
 - **Swagger**
 - WADL
 - WSDL 2.0
 - HTML for humans
 - ...

Summary

Summary

- Remote communication
- Problems introduced by remote communication
- Possible solutions
- Technologies for remote communication
 - SOAP
 - mapping to .NET: Windows Communication Foundation (WCF)
 - mapping to Java: JAX-WS
 - REST
 - mapping to .NET: Windows Communication Foundation (WCF)
 - mapping to Java: JAX-RS

Concurrent and Distributed Patterns

Objektumorientált szoftvertervezés

Object-oriented software design

Dr. Balázs Simon

BME, IIT

Outline

- Concurrent and distributed problems
- Possible solutions:
 - Synchronization patterns
 - Context patterns
 - Request/event handling patterns
 - Concurrency patterns

Concurrent and distributed problems

Problems of concurrency

- Mutable shared state
- Race conditions
- Synchronization
- Dead-locks
- Starvation
- Can't be exactly reproduced
- Can't be tested
- High complexity

Problems of distributed systems

- Heterogeneity
 - different programming languages
- Transparency
 - other side should not seem to be remote or distributed
- Synchronous and asynchronous processing
- Memory management
- Caching
- Replicas and consistency
- Latency
- Scalability
- Concurrency
- Security
- Network failures
- Hard to monitor
- Hard to debug

Synchronization patterns

Synchronization patterns

- Synchronizing the execution of threads
- Patterns:
 - Critical sections: operations appear atomic
 - Atomic operations
 - Scoped locking
 - Balking: wait until the object is in the appropriate state
 - Balking design pattern
 - Double-checked locking
 - Guarded suspension
 - Signaling: notifying other threads
 - Monitor object
 - Mutex
 - Semaphore
 - AutoResetEvent
 - ManualResetEvent
 - Readers-writer lock
 - Public interface: efficiency and recursion
 - Strategized locking
 - Thread-safe interface

Synchronization patterns

Critical sections: operations appear atomic

Atomic operation

- Appears to the rest of the system to occur instantaneously
- Guarantee of isolation from concurrent processes
- C#: System.Threading namespace

```
// ++counter;  
Interlocked.Increment(ref counter);  
// tmp = obj; obj = value; return tmp;  
tmp = Interlocked.Exchange(ref obj, value);  
// tmp = obj; if (obj == coparand) { obj = value; } return tmp;  
tmp = Interlocked.CompareExchange(ref obj, value, comparand);
```

- Java: java.util.concurrent.atomic package

```
// ++counter;  
AtomicInteger counter = new AtomicInteger(0);  
counter.incrementAndGet();  
// tmp = obj; obj = value; return tmp;  
AtomicReference<Object> obj = new AtomicReference<Object>();  
tmp = obj.set(value);  
// if (obj == coparand) { obj = value; return true; } else { return false; }  
obj.compareAndSet(comparand, value);
```

Scoped locking

- Defining critical sections
- Grouping multiple operations as if they were atomic
- Scoped locks in .NET and Java are reentrant: recursive calls on the same thread do not cause dead-lock
- C#: lock keyword

```
lock (obj)
{
    // ...
}
```

- Java: synchronized keyword

```
synchronized (obj) {
    // ...
}
```

Synchronization patterns

Balking: wait until the object is in the appropriate state

Balking

- If a method is invoked when the object is in an inappropriate state, then the method will return without doing anything or wait until the state is appropriate
- Balking patterns:
 - Balking design pattern: perform an operation only in a given state, otherwise return without doing anything
 - e.g. the operation is already in progress
 - Double-checked locking optimization: acquire a lock only if necessary
 - e.g. lazy initialization of a singleton object
 - Guarded suspension: wait until a lock is acquired and a precondition is met
 - e.g. wait until there is something to process

Balking design pattern

- Return immediately if the state is inappropriate:
 - e.g. job is already in progress

```
public class Example {  
    private bool jobInProgress = false;  
  
    public void ExecuteJob() {  
        lock (this) {  
            if (jobInProgress) {  
                return;  
            }  
            jobInProgress = true;  
        }  
  
        // Code to execute job goes here  
        // ...  
  
        lock (this) {  
            jobInProgress = false;  
        }  
    }  
}
```

Double-checked locking

- Problem of implementing a singleton:

```
public class Singleton
{
    private static Singleton singleton = null;

    private Singleton() { }

    public static Singleton GetInstance()
    {
        if (singleton == null)
        {
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

Problem:

two threads may enter the “if” simultaneously:
singleton is created twice

Double-checked locking

- A possible solution:

```
public class Singleton
{
    private static object myLock = new object();
    private static Singleton singleton = null;

    private Singleton() { }

    public static Singleton GetInstance()
    {
        lock (myLock) // Problem:
        {
            if (singleton == null)
            {
                singleton = new Singleton();
            }
            return singleton;
        }
    }
}
```

lock/synchronized is expensive,
and it is executed every time the method is called

Double-checked locking

- Double-checked locking (OK in .NET 2.0+, but not in .NET 1 or Java):

```
public class Singleton
{
    private static object myLock = new object();
    private static Singleton singleton = null;

    private Singleton() {}

    public static Singleton GetInstance()
    {
        if (singleton == null)
        { // 1st check
            lock (myLock)
            {
                if (singleton == null)
                { // 2nd (double) check
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

More efficient:
lock/synchronized is called
only if necessary

Problem in Java and .NET 1:
pointer is set before the constructor
is finished executing, the second thread can
return a partially constructed object

Double-checked locking

- Solution (OK in .NET 1+ and Java 5+, but not Java 4-): volatile

```
public class Singleton
{
    private static object myLock = new object();
    private volatile static Singleton singleton = null;

    private Singleton() {}

    public static Singleton GetInstance()
    {
        if (singleton == null)
        { // 1st check
            lock (myLock)
            {
                if (singleton == null)
                { // 2nd (double) check
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
```

Problem in Java 4-:
same as before, since the semantics of volatile
was only corrected in Java 5

Double-checked locking

- Very hard to implement it correctly
 - due to compiler optimizations
 - due to processor optimizations
- .NET 2.0+:
 - works without volatile, since the lock keyword enforces correct execution order
- .NET 1, Java 5+:
 - works only with volatile
- Java 4-:
 - There is no correct solution for the double-checked locking, don't use it!

Avoiding double-checked locking

- Using static initialization (always correct in .NET and Java):

```
public class Singleton
{
    private static readonly Singleton singleton = new Singleton();

    private Singleton() {}

    public static Singleton GetInstance()
    {
        return singleton;
    }
}
```

Problems:

- it is not lazily initialized
- static initialization order is not deterministic
- cannot be used for non-static (non-singleton) cases

Avoiding double-checked locking

- Using static lazy initialization (always correct in .NET and Java):

```
public class Singleton
{
    // Lazy initialization:
    private class Holder ← should be a static class in Java
    {
        public static readonly Singleton singleton = new Singleton();
    }

    private Singleton() {}

    public static Singleton GetInstance()
    {
        return Holder.singleton;
    }
}
```

Problem:

- cannot be used for non-static (non-singleton) cases

Guarded suspension

- Waiting until a lock is acquired and a precondition is met:

```
public void operationWithPrecondition() {  
    synchronized (lock) {  
        while (!preCondition) {  
            try {  
                lock.wait();  
            } catch (InterruptedException e) {}  
        }  
        // ...  
    }  
}
```

Must be a while, not an if:
when the thread is awoken or interrupted,
the pre-condition may still not be met

The wait() exits from the synchronized block:
other threads can enter the synchronized block
(Never create an empty while loop!
It is busy waiting and consumes very high CPU!)

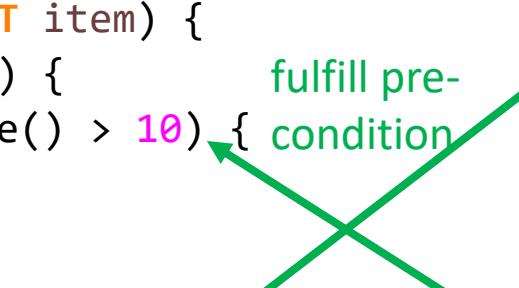
```
public void fulfillPrecondition() {  
    synchronized (lock) {  
        preCondition = true;  
        lock.notify();  
    }  
}
```

It is not deterministic which
thread is awoken

Guarded suspension

■ Example: FIFO

```
public class Fifo<T> {  
    private Object lock = new Object();  
    private ArrayList<T> items =  
        new ArrayList<>();  
  
    public void enqueue(T item) {  
        synchronized (lock) {  
            while (items.size() > 10) {  
                try {  
                    lock.wait();  
                }  
                catch (InterruptedException e)  
                {}  
            }  
            items.add(item);  
            lock.notifyAll();  
        }  
    }  
  
    public T dequeue() {  
        T result;  
        synchronized (lock) {  
            while (items.size() == 0) {  
                try {  
                    lock.wait();  
                }  
                catch (InterruptedException e)  
                {}  
            }  
            result = items.get(0);  
            items.remove(0);  
            lock.notifyAll();  
        }  
        // ...  
    }  
}
```



A large green X is drawn across the code, spanning from the end of the enqueue method to the start of the dequeue method. A green arrow points from the text "fulfill precondition" to the condition in the enqueue loop.

Synchronization patterns

Signaling: notifying other threads

Signaling

- Threads notify other threads
 - e.g. in the case of guarded suspension
- Cases:
 - Monitor object: mutual exclusion + signaling other threads that their waiting condition has been met
 - Mutex: only one thread can enter (mutual exclusion)
 - Semaphore: only a given number of threads can enter
 - ManualResetEvent: allow multiple threads to continue after an operation is done
 - AutoResetEvent: allow a single thread to continue after an operation is done
 - Readers-writer lock: allow multiple reads but only a single write
- C#: common base class for signals is
`System.Threading.WaitHandle`
- Java: no common base class for signals

Monitor object

- Allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true
- Provides a mechanism for signaling other threads that their condition has been met
- Java: every object is a Monitor object
 - mutual exclusion: synchronized keyword with the object as parameter
 - signaling: wait(), notify(), notifyAll()
- C#: System.Threading.Monitor static class
 - provides static utility functions for treating regular objects as Monitor objects
 - mutual exclusion: Enter(object), Exit(object)
 - signaling: Wait(object), Pulse(object), PulseAll(object)

Monitor object: Java

- Example: FIFO (same as the guarded suspension example)

```
public class Fifo<T> {                                in this example this is the monitor object
    private Object lock = new Object();
    private ArrayList<T> items =
        new ArrayList<>();

    public void enqueue(T item) {
        synchronized (lock) {
            while (items.size() > 10) {                fulfill pre-
                try {                                     condition
                    lock.wait();}
                catch (InterruptedException e) {}}
            items.add(item);
            lock.notifyAll();}
    }

    public T dequeue() {
        T result;
        synchronized (lock) {
            while (items.size() == 0) {}
            try {
                lock.wait();
            } catch (InterruptedException e) {}
            result = items.get(0);
            items.remove(0);
            lock.notifyAll();}
        return result;
    }

    // ...
}
```

The diagram illustrates the interaction between the `enqueue` and `dequeue` methods. It shows the flow of control and the use of the `lock` object as a monitor.

- enqueue Method:** This method uses the `lock` object for synchronization. It contains a loop that checks if the size of the `items` list is greater than 10. If it is, it calls `lock.wait()`. After the loop, it adds the `item` to the list and then calls `lock.notifyAll()`.
- dequeue Method:** This method also uses the `lock` object for synchronization. It contains a loop that checks if the size of the `items` list is 0. If it is, it calls `lock.wait()`. After the loop, it retrieves the first item from the list, removes it, and then calls `lock.notifyAll()`.
- Shared Resource:** The `lock` variable is highlighted in both methods, indicating that it is a shared monitor object used for mutual exclusion.

Monitor object: C#

- Example: FIFO (same as the guarded suspension example)

```
public class Fifo<T>
{
    private object obj = new object();  
    private List<T> items = new List<T>();  
  
    public void Enqueue(T item)  
    {  
        Monitor.Enter(obj);  
        try  
        {  
            while (items.Count > 10)  
            {  
                Monitor.Wait(obj);  
            }  
            items.Add(item);  
            Monitor.PulseAll(obj);  
        }  
        finally  
        {  
            Monitor.Exit(obj);  
        }  
    }  
  
    public T Dequeue()  
    {  
        Monitor.Enter(obj);  
        try  
        {  
            T result;  
            while (items.Count == 0)  
            {  
                Monitor.Wait(obj);  
            }  
            result = items[0];  
            items.RemoveAt(0);  
            Monitor.PulseAll(obj);  
            return result;  
        }  
        finally  
        {  
            Monitor.Exit(obj);  
        }  
    }  
}
```

in this example this is the monitor object

fulfill pre-condition

Semaphore

- Used to control access to a pool of resources
 - we have k resources and k keys
 - anyone who has a key can access a resource
 - example: k toilets with identical locks and keys
- A semaphore is decremented each time a thread enters the semaphore
- A semaphore is incremented when a thread releases the semaphore
- A semaphore blocks if the counter reaches zero and the thread tries to enter
- C#: System.Threading.Semaphore

```
Semaphore semaphore = new Semaphore(initialCount, maximumCount);
// Decrease counter:
semaphore.WaitOne();
// Increase counter:
semaphore.Release();
```

- Java: java.util.concurrent.Semaphore

```
Semaphore semaphore = new Semaphore(MAX_COUNT);
// Decrease counter:
semaphore.acquire();
// Increase counter:
semaphore.release();
```

Mutex

- Synchronization primitive that grants exclusive access to the shared resource to only one thread
- If a thread acquires a mutex, the second thread that wants to acquire that mutex is suspended until the first thread releases the mutex
- Example: a mutex is a key to a toilet, one person can occupy the toilet at a time
- Mutex is the same as Semaphore with counter = 1
- C#: System.Threading.Mutex class

```
Mutex mutex = new Mutex();  
// Acquire:  
mutex.WaitOne();  
// Release:  
mutex.ReleaseMutex();
```

- Java: java.util.concurrent.Semaphore with max. count = 1

Manual reset event

- Allow multiple threads to continue after an operation is done
- It is like a door, which needs to be closed (reset) manually
 - people can go through as long as the door is open
- Two states:
 - signaled (set): threads are allowed to continue
 - non-signaled (reset): threads are blocked
- C#: System.Threading.ManualResetEvent
 - Set(): make it signaled
 - Reset(): make it non-signaled
 - WaitOne():
 - returns immediately and allows the thread to continue if the event is signaled
 - blocks if the event is non-signaled
- Java: no such solution, but can be implemented easily

Implementation of manual reset event

```
public class ManualResetEvent {  
  
    private final Object monitor = new Object();  
    private volatile boolean signaled = false;  
  
    public ManualResetEvent(boolean signaled) {  
        this.signaled = signaled;  
    }  
  
    public void set() {  
        synchronized (monitor) {  
            signaled = true;  
            monitor.notifyAll();  
        }  
    }  
  
    public void reset() {  
        synchronized (monitor) { // required only in Java 4-  
            signaled = false;  
        }  
    }  
}
```

Implementation of manual reset event

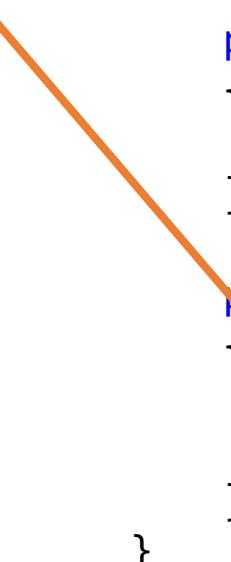
```
public void waitOne() {  
    synchronized (monitor) {  
        while (!signaled) {  
            try {  
                monitor.wait();  
            } catch (InterruptedException e) {  
                // nop  
            }  
        }  
    }  
}
```

Implementation of manual reset event

```
public boolean waitOne(long timeout) {  
    synchronized (monitor) {  
        long t = System.currentTimeMillis();  
        while (!signaled) {  
            try {  
                monitor.wait(timeout);  
            } catch (InterruptedException e) {  
                // nop  
            }  
            // Check for timeout  
            if (System.currentTimeMillis() - t >= timeout) {  
                break;  
            }  
        }  
        return signaled;  
    }  
}
```

Manual reset event example

```
// Thread 1:  
public class Downloader  
{  
    public ManualResetEvent Downloaded { get; }  
  
    public Downloader()  
    {  
        this.Downloaded = new ManualResetEvent(false);  
    }  
  
    public void Download()  
    {  
        this.Downloaded.Reset();  
        // ... looong operation ...  
        this.Downloaded.Set();  
    }  
}  
  
// Thread 2:  
public class FileOpener  
{  
    private Downloader downloader;  
  
    public FileOpener(Downloader downloader)  
    {  
        this.downloader = downloader;  
    }  
  
    public void OpenFile()  
    {  
        this.downloader.Downloaded.WaitOne();  
        // ... open downloaded file ...  
    }  
}
```



Auto reset event

- Allow a single thread to continue after an operation is done
- It is like a tollbooth
 - allows one car to go by and automatically closes before the next one can get through
- Same as a Manual reset event, except that `WaitOne()` automatically calls `Reset()`, so other threads won't be able to continue
- C#: `System.Threading.ManualResetEvent`
 - `Set()`: make it signaled
 - `Reset()`: make it non-signaled
 - `WaitOne()`:
 - returns immediately and allows the thread to continue if the event is signaled
 - blocks if the event is non-signaled
 - calls `Reset()` before returning, so other threads will be blocked
- Java: no such solution, but can be implemented easily

Implementation of auto reset event

```
public class AutoResetEvent {  
  
    private final Object monitor = new Object();  
    private volatile boolean signaled = false;  
  
    public AutoResetEvent(boolean signaled) {  
        this.signaled = signaled;  
    }  
  
    public void set() {  
        synchronized (monitor) {  
            signaled = true;  
            monitor.notifyAll();  
        }  
    }  
  
    public void reset() {  
        synchronized (monitor) { // required only in Java 4-  
            signaled = false;  
        }  
    }  
}
```

Implementation of auto reset event

```
public void waitOne() {  
    synchronized (monitor) {  
        while (!signaled) {  
            try {  
                monitor.wait();  
            } catch (InterruptedException e) {  
                // nop  
            }  
        }  
        signaled = false;  
    }  
}
```

Implementation of auto reset event

```
public boolean waitOne(long timeout) {  
    synchronized (monitor) {  
        try {  
            long t = System.currentTimeMillis();  
            while (!signaled) {  
                try {  
                    monitor.wait(timeout);  
                } catch (InterruptedException e) {  
                    // nop  
                }  
                // Check for timeout  
                if (System.currentTimeMillis() - t >= timeout) {  
                    break;  
                }  
            }  
            return signaled;  
        } finally {  
            signaled = false;  
        }  
    }  
}
```

Auto reset event example

```
public class PrinterSpooler
{
    private AutoResetEvent printerGuard;

    public PrinterSpooler()
    {
        printerGuard = new AutoResetEvent(true);
    }

    public void Print(PrintJob printJob)
    {
        this.printerGuard.WaitOne();
        // If we reach here, we have sole access to the printer.
        // ... print the job
        this.printerGuard.Set();
    }
}
```

ManualResetEvent and AutoResetEvent

- Warning! Don't do this:

```
public class Periodic
{
    public AutoResetEvent Guard { get; }

    public void PeriodicSignal()
    {
        while (true)
        {
            this.Guard.Set(); // Wrong! May not signal the other thread!
            this.Guard.Reset(); // Because Reset() may run too soon!
            Thread.Sleep(1000);
        }
    }
}
```

- For generating a periodic signal use Monitor instead:

- notify() or Pulse() inside the loop (instead of Set()-Reset())
- wait() or Wait() from other threads (instead of WaitOne())

Readers-writer lock

- Accessing a single resource
- Readers can run in parallel as long as the resource is not written
- Writers get exclusive access, only a single writer can run at a time, and also readers are blocked until the writer is finished
- When the writer tries to acquire a lock:
 - existing readers will be allowed to finish
 - the writer is blocked until then
 - new readers will be blocked
 - when all existing readers are finished, the writer gets the lock
 - when the writer is finished, readers can run again
- Be careful with recursion! It may cause dead-locks!
- C#: System.Threading.ReaderWriterLockSlim
- Java: java.util.concurrent.locks.ReentrantReadWriteLock

Synchronization patterns

Public interface: efficiency and recursion

Strategized locking

- Run efficiently in a variety of different concurrency architectures
 - single threaded architecture: no locks are necessary
 - multi-threaded architecture: locks are necessary
- If we always use locks: we penalize single threaded applications which could run more efficiently without locks
- Idea: make the locking mechanism dynamically replaceable
- Realization:
 - Implement a null-object version of the Critical section, Monitor object and other signaling patterns
 - these implementations are empty, they do not lock
 - Use the null-object versions in single threaded applications
 - Use the original versions in multi-threaded applications

Thread-safe interface

- Problem:
 - in some programming languages or libraries locks are not reentrant, i.e. they cause dead-lock in a recursion
 - Readers-writer lock is also dangerous, it can cause dead-lock
- Solution:
 - the public methods of an object should use locks
 - they should call protected/private methods which do not use locks
 - this way protected/private methods can be recursive and more efficient
 - no danger of dead-lock from recursion

Context patterns

Context patterns

- Provide context specific information for threads
- Patterns:
 - Global context
 - Thread-local storage
 - Thread-local context

Global context

- Provides a globally accessible context (execution environment)
- Useful if there is no dependency injection and we do not want to pass the context to every method
- It is like a static singleton, but available only in a given scope
 - careful: other threads may run outside the scope! In a multi-threaded application use Thread-local context!
- Example:

```
public class SomeClass {  
    public void SomeMethod() {  
        // Access the value stored in GlobalContext: "Hello"  
        string currentValue = GlobalContext.Current.SomeValue;  
    }  
}  
  
public class SomeProgram {  
    public static void Main(string[] args) {  
        // Make GlobalContext available only in this scope:  
        using (var scope = new GlobalContextScope("Hello")) {  
            SomeClass cls = new SomeClass();  
            cls.SomeMethod();  
        }  
    }  
}
```

Implementation of GlobalContext

```
public class GlobalContext
{
    // Instantiated only by GlobalContextScope:
    internal GlobalContext(string value)
    {
        this.SomeValue = value;
    }

    // An option/value available in the context:
    public string SomeValue { get; }

    // Gets the current context, set only by GlobalContextScope:
    public static GlobalContext Current { get; internal set; }
}
```

Implementation of GlobalContextScope

```
public class GlobalContextScope : IDisposable
{
    private static object lockObj = new object();

    // Set the GlobalContext if it is not yet set:
    public GlobalContextScope(string value) {
        lock (lockObj) {
            if (GlobalContext.Current != null) {
                throw new InvalidOperationException(
                    "The global context is already set.");
            }
            GlobalContext.Current = new GlobalContext(value);
        }
    }

    // Remove the GlobalContext if the scope is ended:
    public void Dispose() {
        lock(lockObj) {
            GlobalContext.Current = null;
        }
    }
}
```

Thread-local storage

- Static fields in .NET and Java are specific to a class across all objects and all threads
- Thread-local storage is usually a static field that stores thread-specific value
 - same value across all objects
 - but a different value for each thread
 - acts like a Dictionary/HashMap where the key is the thread
- Thread-local instance (i.e. non-static) fields are rarely used
- C#: System.Threading.ThreadLocal<T>
- Java: java.lang.ThreadLocal<T>

Thread-local context

- Provides a globally accessible thread-specific context (execution environment)
- Useful if there is no dependency injection and we do not want to pass the context to every method
- Similar to a Global context, but implemented with thread-local storage
- Can be used on the server side to provide execution context for worker threads
 - e.g. WCF OperationContext

Thread-local context example

```
public class SomeClass {
    public void SomeMethod() {
        // Access the value stored in ThreadLocalContext: "Hello"
        string currentValue = ThreadLocalContext.Current.SomeValue;
    }
}
public class SomeThread {
    public void Run() {
        // Make ThreadLocalContext available only in this scope:
        using (var scope = new ThreadLocalContextScope("Hello")) {
            SomeClass cls = new SomeClass();
            cls.SomeMethod();
        }
    }
}
```

Implementation of thread-local context

```
public class ThreadLocalContext
{
    // Instantiated only by ThreadLocalContextScope:
    internal ThreadLocalContext(string value)
    {
        this.SomeValue = value;
    }

    // An option/value available in the context:
    public string SomeValue { get; }

    private static ThreadLocal<ThreadLocalContext> current =
        new ThreadLocal<ThreadLocalContext>();

    // Gets the current context, set only by ThreadLocalContextScope:
    public static ThreadLocalContext Current
    {
        get { return ThreadLocalContext.current.Value; }
        internal set { ThreadLocalContext.current.Value = value; }
    }
}
```

Implementation of thread-local context scope

```
// Create a separate scope for each thread.  
// Locking is not necessary any more, since only the current thread  
// has access, and hence there is no shared state between threads:  
public class ThreadLocalContextScope : IDisposable  
{  
    // Set the ThreadLocalContext if it is not yet set:  
    public ThreadLocalContextScope(string value)  
    {  
        if (ThreadLocalContext.Current != null)  
        {  
            throw new InvalidOperationException(  
                "The thread-local context is already set.");  
        }  
        ThreadLocalContext.Current = new ThreadLocalContext(value);  
    }  
  
    // Remove the ThreadLocalContext if the scope is ended:  
    public void Dispose()  
    {  
        ThreadLocalContext.Current = null;  
    }  
}
```

Request/event handling patterns

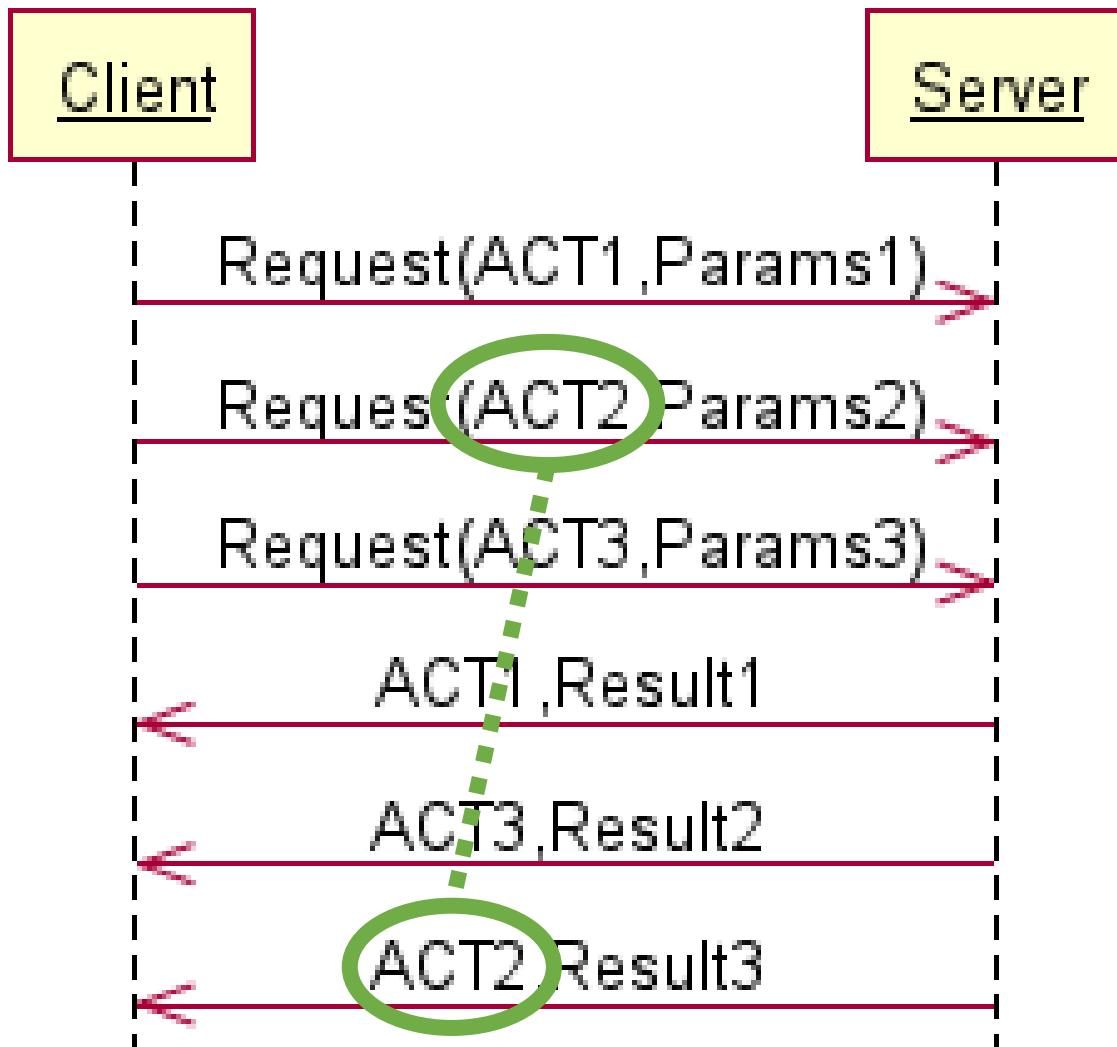
Request/event handling patterns

- Handling synchronous and asynchronous requests
- Input-output handling:
 - blocking IO: read/write waits until completed, calling thread blocks
 - non-blocking, synchronous IO: read/write returns immediately, either with the data read/written or with a signal that the IO operation could not be completed
 - non-blocking, asynchronous IO: read/write returns immediately, operation is started on a background thread, the caller is notified when the operation is ready
- Patterns:
 - Asynchronous completion token
 - Cancellation token
 - Future/Task/Deferred (async-await)
 - Reactor: non-blocking, synchronous IO
 - Proactor: non-blocking, asynchronous IO

Asynchronous completion token (ACT)

- Problem: client calls multiple asynchronous operations on the server and receives responses for them but not necessarily in order
- The ACT pattern allows efficient demultiplexing responses of asynchronous operations
- ACT pattern:
 - the ACT is usually an identifier
 - client passes the ACT in asynchronous requests
 - server returns the original ACT in asynchronous responses
 - based on the returned ACT the client can identify which response is for which request

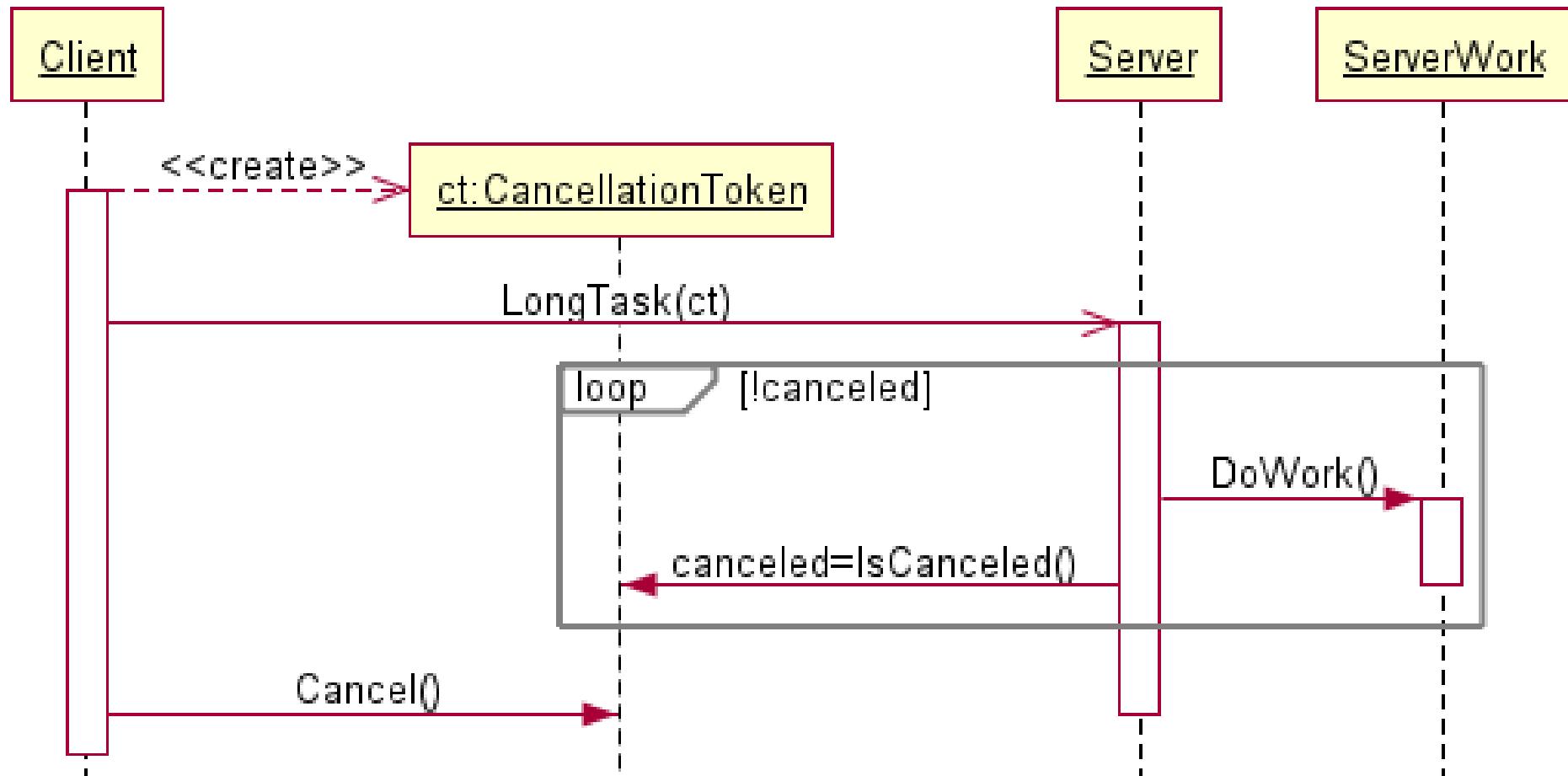
Asynchronous completion token (ACT)



Cancellation token

- Problem:
 - the client calls a long running asynchronous operation on the server frequently
 - the long running operation may not finish before it is called again
 - the result of the old operation becomes irrelevant when the operation is called again
 - the client needs a way to cancel the old operation
 - example:
 - client is the IDE editor, while the programmer is editing source code
 - server is the IDE compiler, which is executed regularly
- Cancellation token pattern:
 - the cancellation token is an object passed to the long running asynchronous operation
 - it can be cancelled by the client
 - the long running asynchronous operation regularly checks the token whether it has been cancelled
 - when the token was cancelled the long running asynchronous operation exits
- .NET:
 - read-only view: System.Threading.CancellationToken class
 - manipulation: System.Threading.CancellationTokenSource class
- Java: no such solution, but can be implemented easily

Cancellation token



Cancellation token example

```
public class Client {
    private CancellationTokenSource tokenSource = new CancellationTokenSource();
    private Server server = new Server();

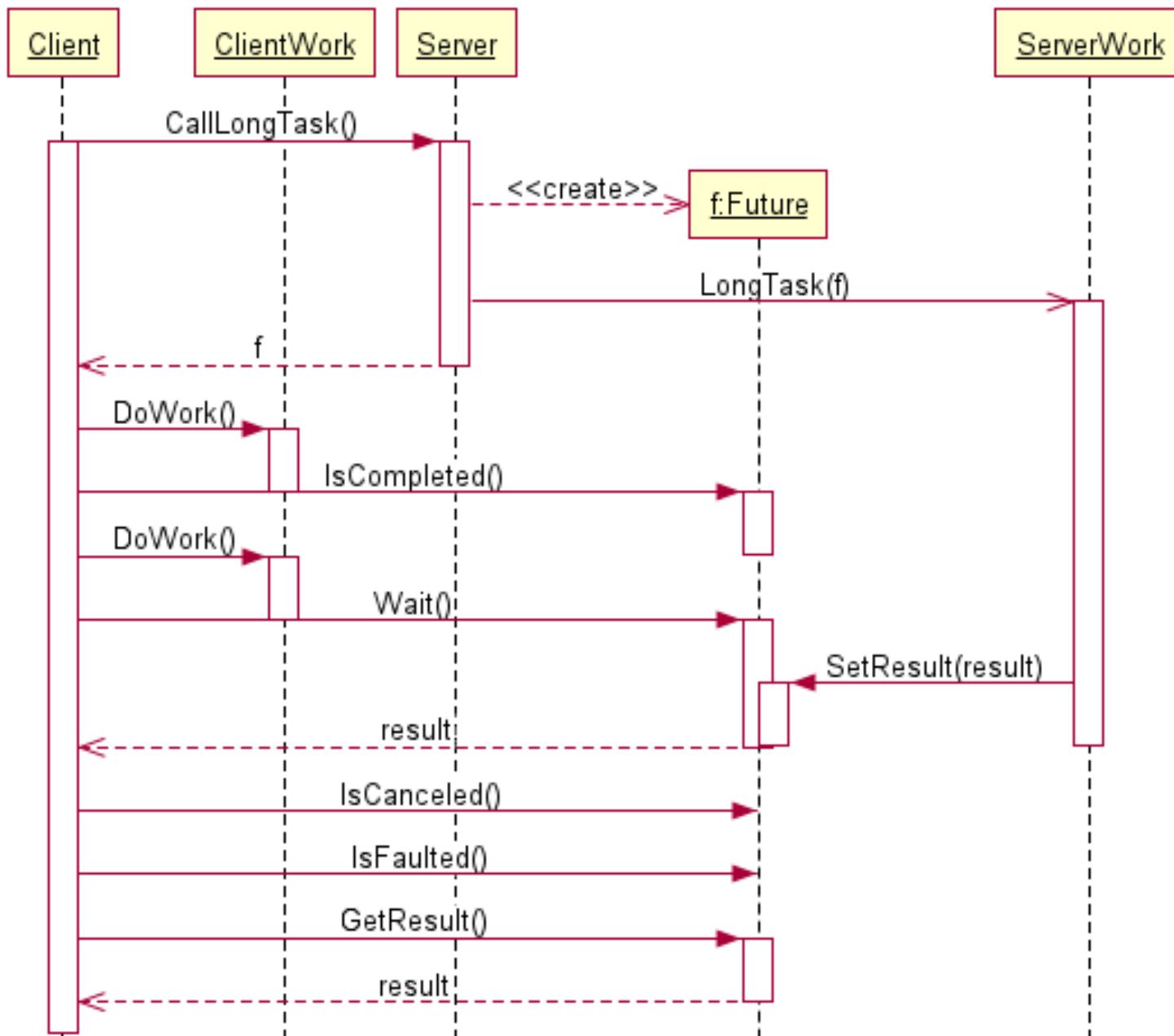
    public void OnEdit() {
        tokenSource.Cancel(); // Cancel previous compilation
        tokenSource = new CancellationTokenSource();
        Task.Run(() => server.Compile(tokenSource.Token));
    }
}

public class Server {
    public Task<bool> Compile(CancellationToken token) {
        this.DoWork();
        // Return if the operation is cancelled:
        if (token.IsCancellationRequested) return Task.FromResult(false);
        this.DoMoreWork(token);
        return Task.FromResult(true);
    }
    private void DoMoreWork(CancellationToken token) {
        this.DoWork();
        // Return from a deep call by throwing OperationCanceledException
        // if the operation is canceled:
        token.ThrowIfCancellationRequested();
        this.DoWork();
    }
}
```

Future/Task/Deferred

- Problem: client starts an asynchronous operation and needs to access or to synchronize for the completion of the result
- Future/Task/Deferred:
 - returned by the asynchronous operation
 - a read-only view of the execution of the asynchronous operation
 - shows whether the operation is running, completed or cancelled
 - allows waiting for the operation to complete and getting the result
- Manipulating the state of a read-only Future/Task/Deferred is usually in a separated class: Promise
 - resolve/bind the result when the operation is completed
 - cancel the operation
- .NET:
 - read-only view: System.Threading.Tasks.Task<T> class
 - manipulation: System.Threading.Tasks.TaskCompletionSource<T> class
 - C# also has built-in language constructs for tasks: async/await
- Java:
 - almost read-only (allows cancellation): java.util.concurrent.Future<T> interface
 - manipulation: java.util.concurrent.FutureTask<T> implementation of Future<T>
 - unfortunately no clear separation of concerns

Future/Task/Deferred



C# example: Task

```
public void CallLongTaskAndDoSomeWork() {
    CancellationTokenSource ct = new CancellationTokenSource();
    Task<CalculationResult> task = this.CallLongTask(ct.Token);
    this.DoWork();
    if (task.IsCompleted) { } // Check if the task is completed
    this.DoWork();
    task.Wait(); // Wait for the task to finish
    CalculationResult result = task.Result; // Get the result
    if (task.IsCanceled) { } // Check if the task is canceled
    if (task.IsFaulted) { } // Check if the task has thrown an exception
}

public Task<CalculationResult> CallLongTask(CancellationToken token) {
    // Cancellation token is optional, included only for the sake of this example:
    Task<CalculationResult> task =
        new Task<CalculationResult>(this.LongTask, token);
    task.Start(); // Start the task in the background
    return task;
}
public CalculationResult LongTask() {
    CalculationResult result = new CalculationResult();
    this.DoWork();
    return result;
}
```

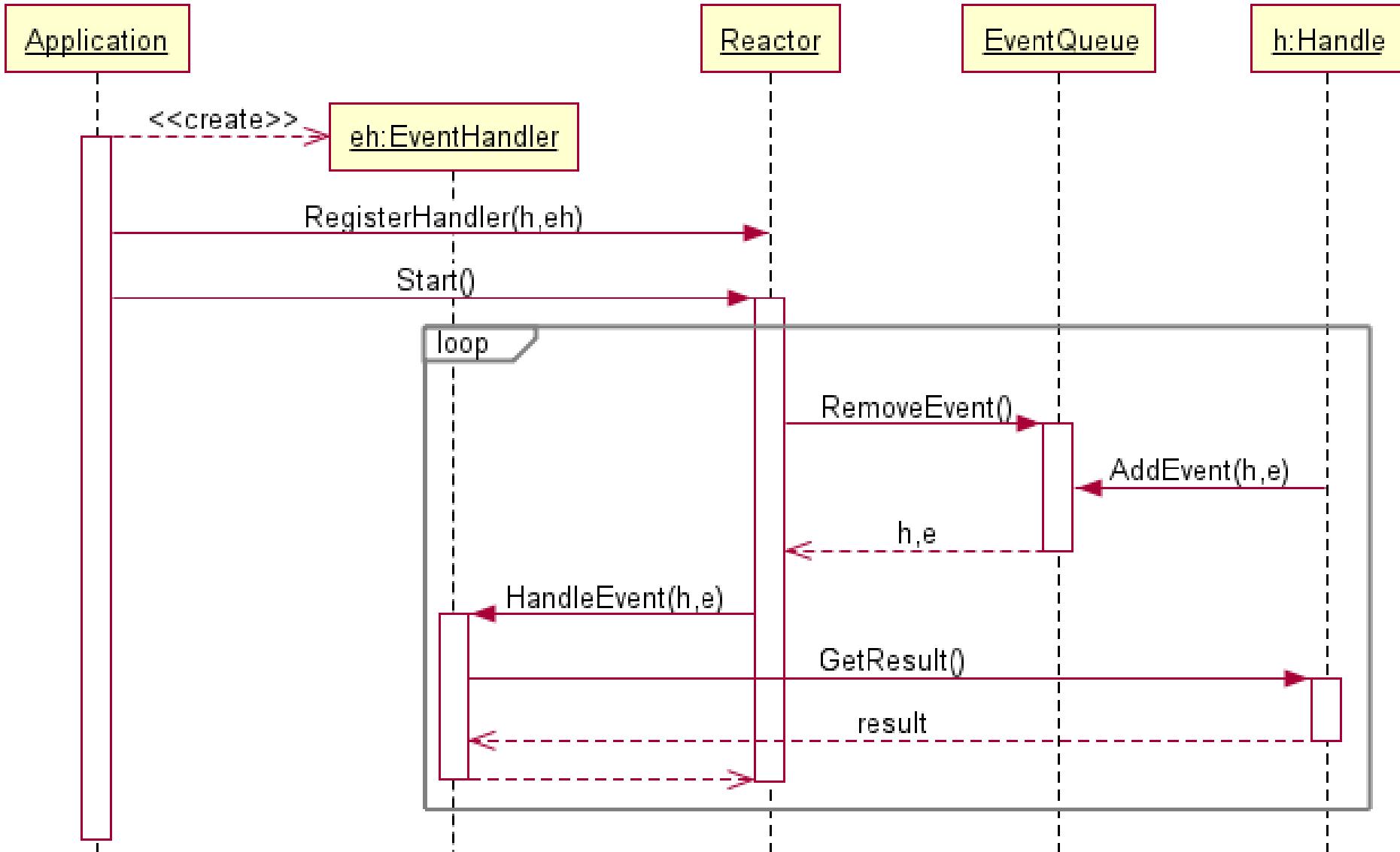
Java example: Future

```
private static final ExecutorService threadpool =
    Executors.newFixedThreadPool(3);
public void callLongTaskAndDoSomeWork() {
    Future<CalculationResult> future = this.callLongTask();
    this.doWork();
    if (future.isDone()) { } // Check if the task is completed
    this.doWork();
    future.cancel(true); // Cancel the task
    this.doWork();
    CalculationResult result = future.get(); // Wait for the task to finish
    if (future.isCancelled()) { } // Check if the task is canceled
    // To check exceptions: override FutureTask
    //      or wrap the task to catch and store exceptions
}
public Future<CalculationResult> callLongTask() {
    FutureTask<CalculationResult> task =
        new FutureTask<CalculationResult>(this::longTask);
    threadpool.execute(task); // Start the task in the background
    return task;
}
public CalculationResult longTask() {
    CalculationResult result = new CalculationResult();
    this.doWork();
    return result;
}
```

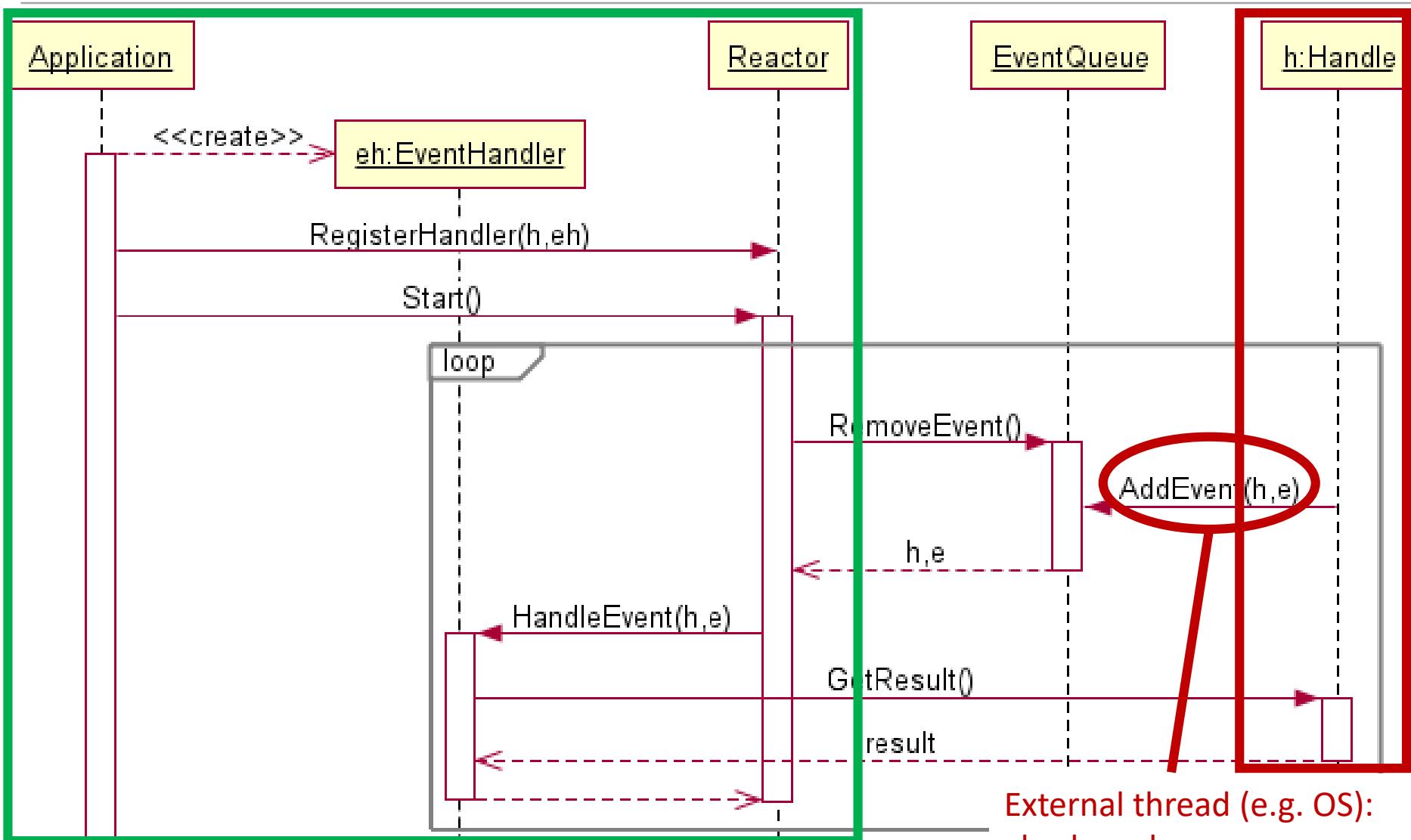
Reactor

- Non-blocking, synchronous IO
 - multiple simultaneous asynchronous (non-blocking) calls
 - synchronous and serial (blocking) processing of the calls
- The pattern consists of the following elements:
 - **Handle:** an entity that has a potentially long running operation, generates and event (asynchronous call) when the result of the operation is ready
 - **Event queue:** stores events generated by the handles
 - **Reactive event handler:** application code, handles (reacts to) the generated events usually by getting the result of the operation (synchronous processing of the call)
 - **Reactor:** allows applications to
 - register or remove event handlers and their associated handles
 - run the event loop to read and demultiplex events from the event queue
 - **Application:** registers event handlers and starts the event loop

Reactor



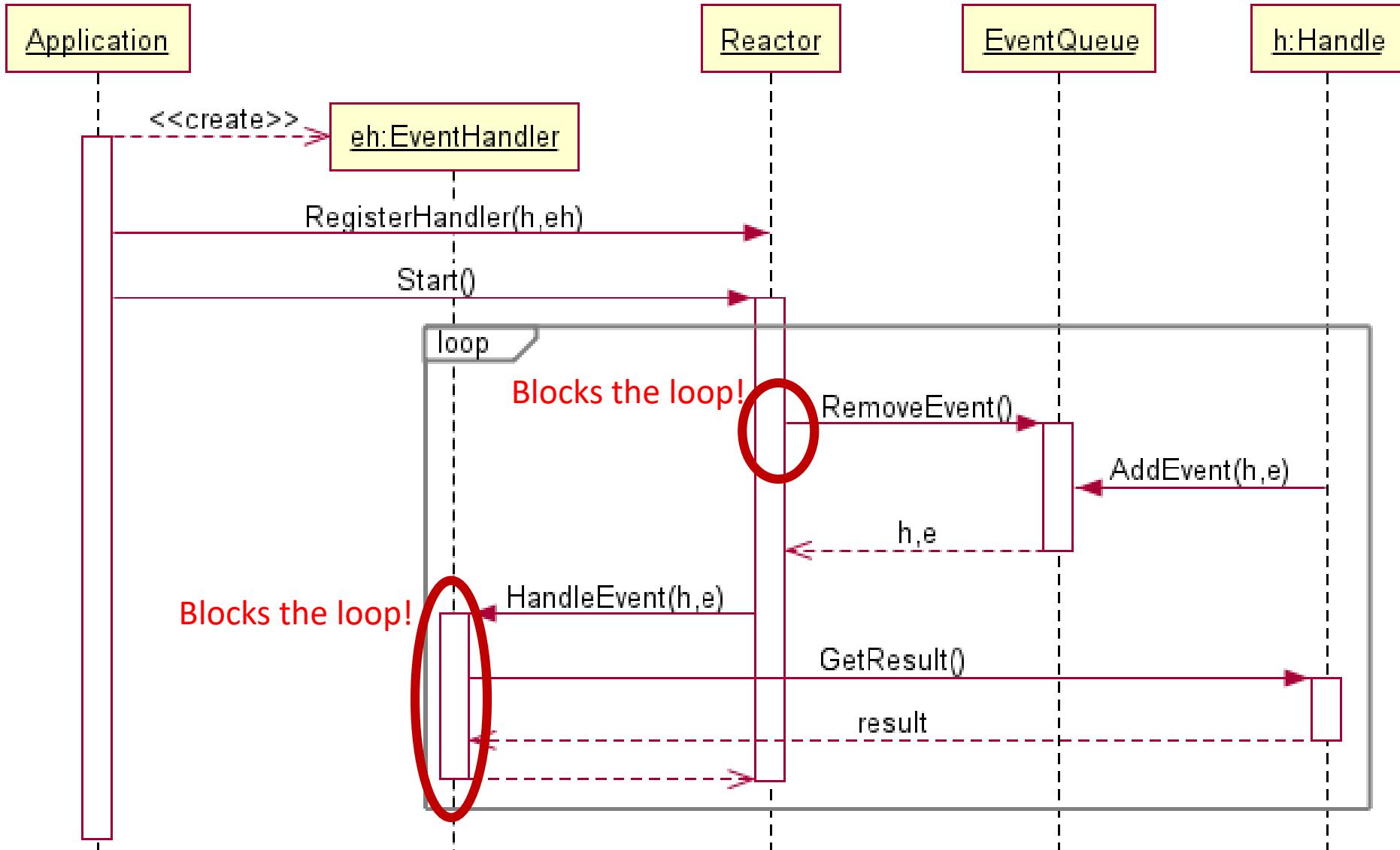
Reactor



Single thread, the application's thread (e.g. UI thread)

External thread (e.g. OS):
keyboard, mouse,
disk read/write,
socket read/write, etc.

Reactor



Reactor

- Reactor's event loop:
 - waits for indication events to occur on its handle set, i.e. events to occur in the event queue
 - when this occurs, the reactor demultiplexes each indication event from the handle on which it occurs to its associated event handler
- Reactor's event loop runs on a single thread, and all event handlers run on this thread, too
 - Don't stop the reactor!
 - Event handlers should be lightweight!
 - don't create long running event handlers
 - don't block inside event handlers
 - don't throw exceptions from event handlers

Reactor examples

- Unix select()
- Windows UI event loop
- Swing listeners
- NodeJS handling a client request

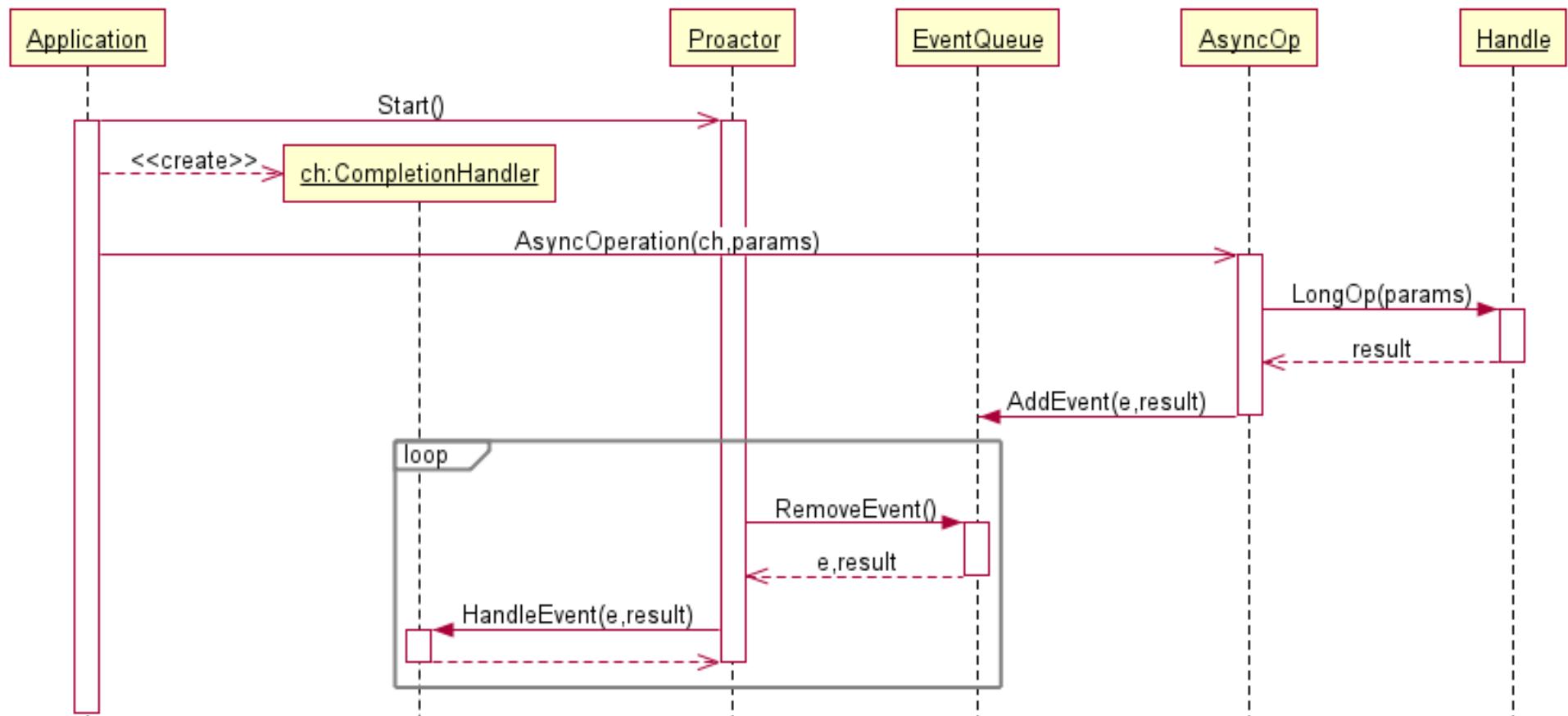
Reactor

- It is the responsibility of a designated component, called Reactor, *not* an application, to wait for indication events synchronously, demultiplex them to associated event handlers that are responsible for processing these events, and then dispatch the appropriate hook method on the event handler
- In particular, a Reactor dispatches event handlers that *react* to the occurrence of a specific event
- Application developers are therefore only responsible for implementing concrete event handlers and can reuse the reactor's demultiplexing and dispatching mechanisms
- Reactor does not scale to support a large number of simultaneous clients and/or long-duration client requests well, because it serializes all event handler processing at the event demultiplexing layer

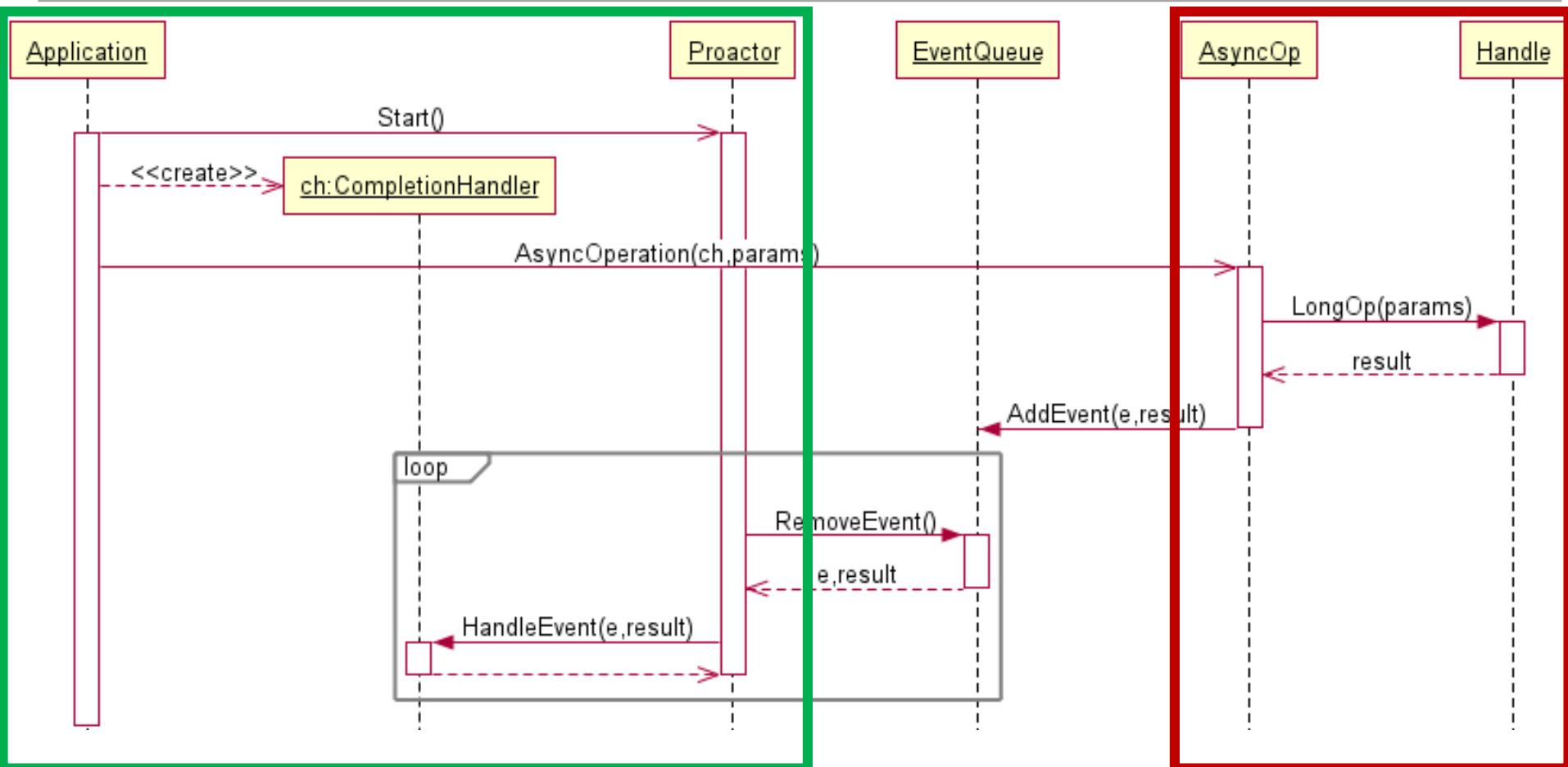
Proactor

- Non-blocking, asynchronous IO
 - asynchronous (non-blocking) calls
 - asynchronous (non-blocking) processing of the calls
- The pattern consists of the following elements:
 - **Handle**: an entity that has a potentially long running operation
 - **Asynchronous operation processor**: executes a long running operation without blocking the client's thread, and generates a completion event with the operation's result when the operation is done
 - **Completion handler**: application code, handles the generated completion events, can proactively invoke other asynchronous operations
 - **Event queue**: stores completion events
 - **Proactor**: provides an event loop for the application
 - **Application**: this is the client that proactively invokes asynchronous operations on an asynchronous operation processor, and often it also provides the completion handler for the operation

Proactor



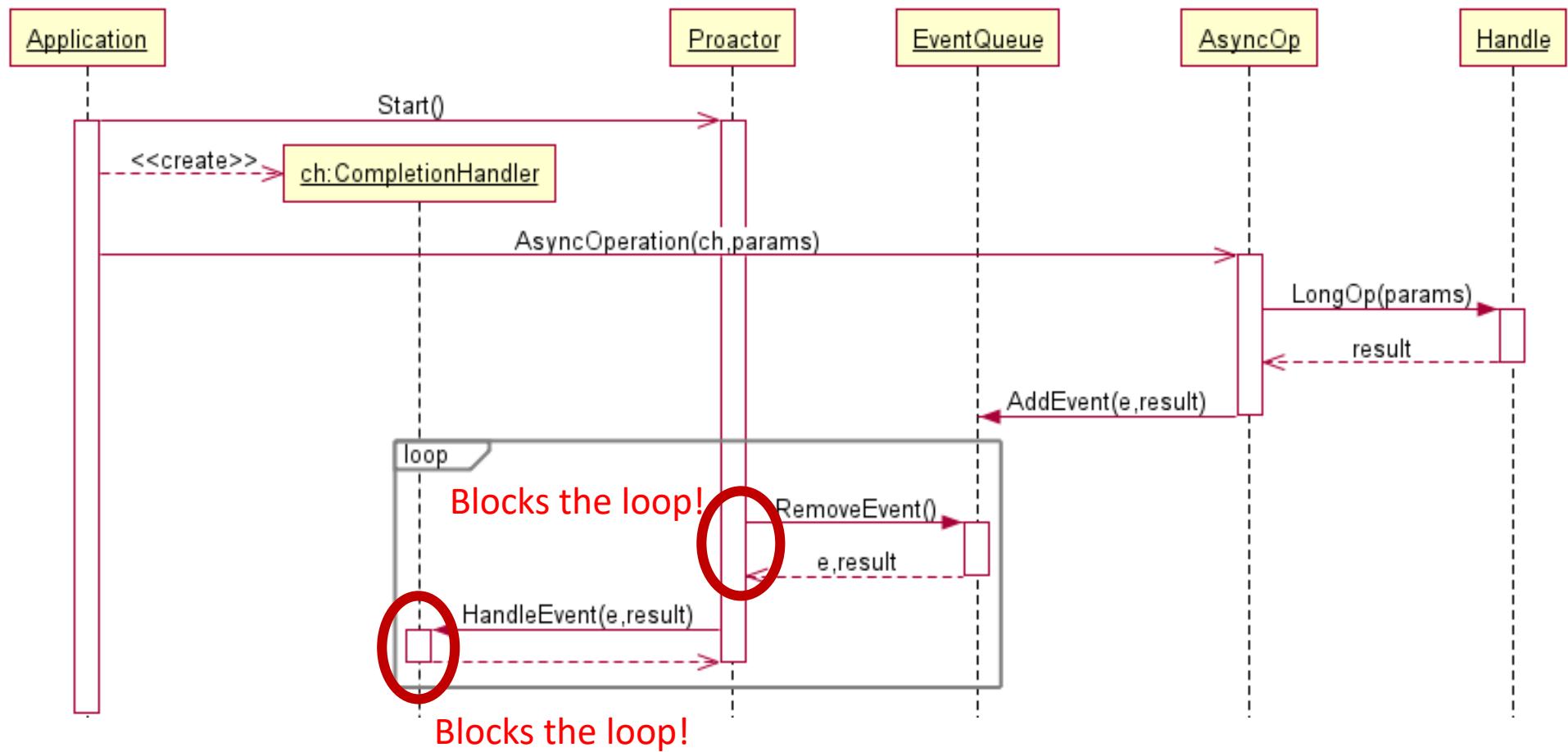
Proactor



Single thread, the application's thread

External thread (e.g. OS thread)
or a background application thread

Proactor



Proactor

- Proactor's event loop:
 - reads completion events from the completion Event queue, then demultiplexes and dispatches these events to the Completion handler
- An Asynchronous operation processor executes the long running operation on a background thread
 - there is a context switching overhead between the threads
- The Application and also the Completion handlers can call other Asynchronous operations
- The Completion handlers run on the same thread as the Proactor, therefore, they block the Proactor's loop
 - Completion handlers should be lightweight
 - if they need to do a lot of processing, they should do that in the background, i.e. they should call other Asynchronous operations

Proactor examples

- C# async-await
- NodeJS calling external IO
 - e.g. mongoose call to MongoDB
- HTTP web servers
 - accessing a file on disk
 - calling a database server
- Browsers calling a web server

Proactor

- In the Proactor pattern, application components – represented by clients and completion handlers – are *proactive* entities
- Unlike the Reactor pattern, which waits passively for indication events to arrive and then reacts, clients and completion handlers in the Proactor pattern instigate the control and data flow within an application by *initiating* one or more asynchronous operation requests proactively on an asynchronous operation processor
- After processing a completion event, a completion handler may initiate new asynchronous operation requests proactively

Concurrency patterns

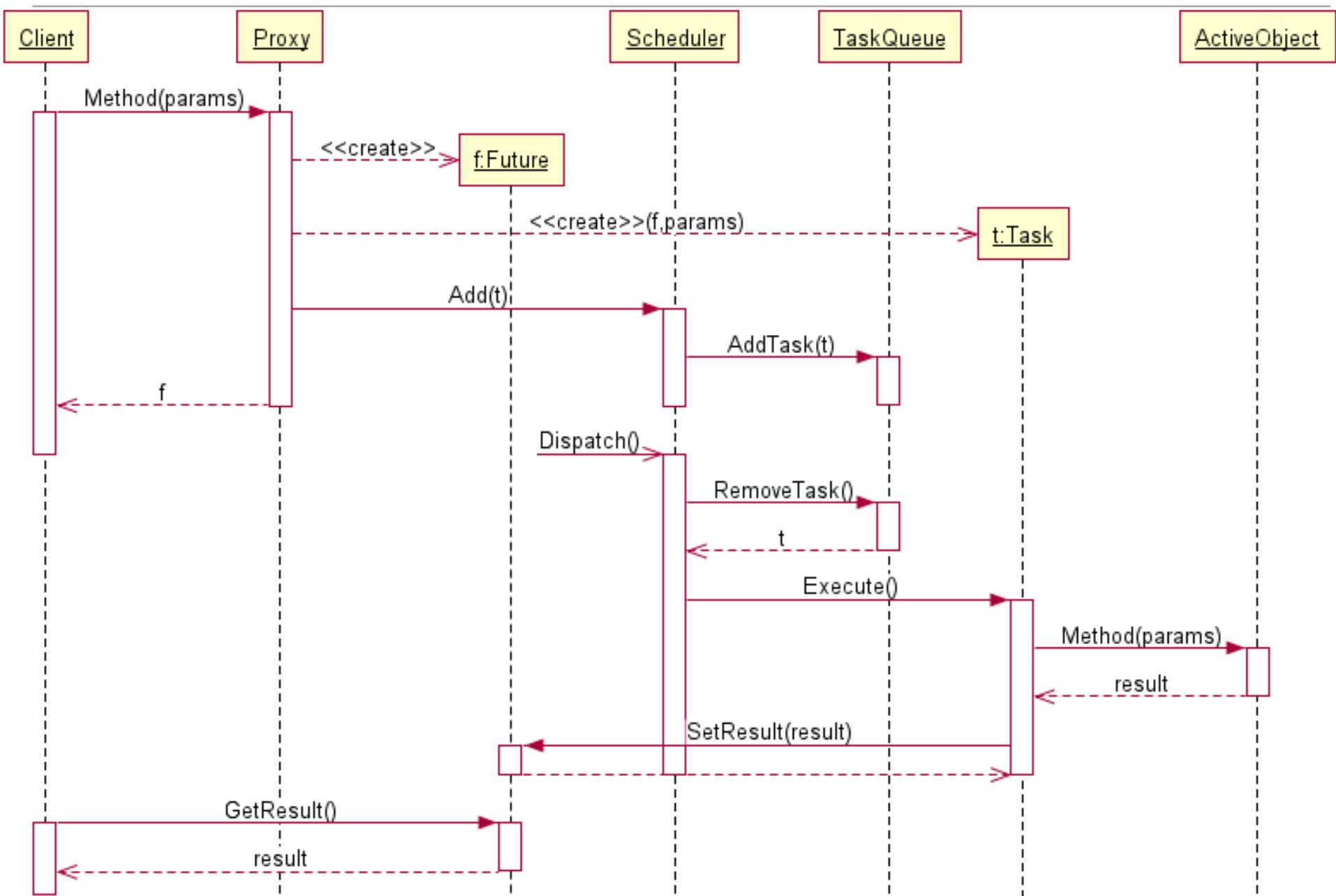
Concurrency patterns

- Processing with multiple threads
- Patterns:
 - Monitor object
 - Active object
 - Half-sync/half-async
 - Leader-followers
 - Scheduler

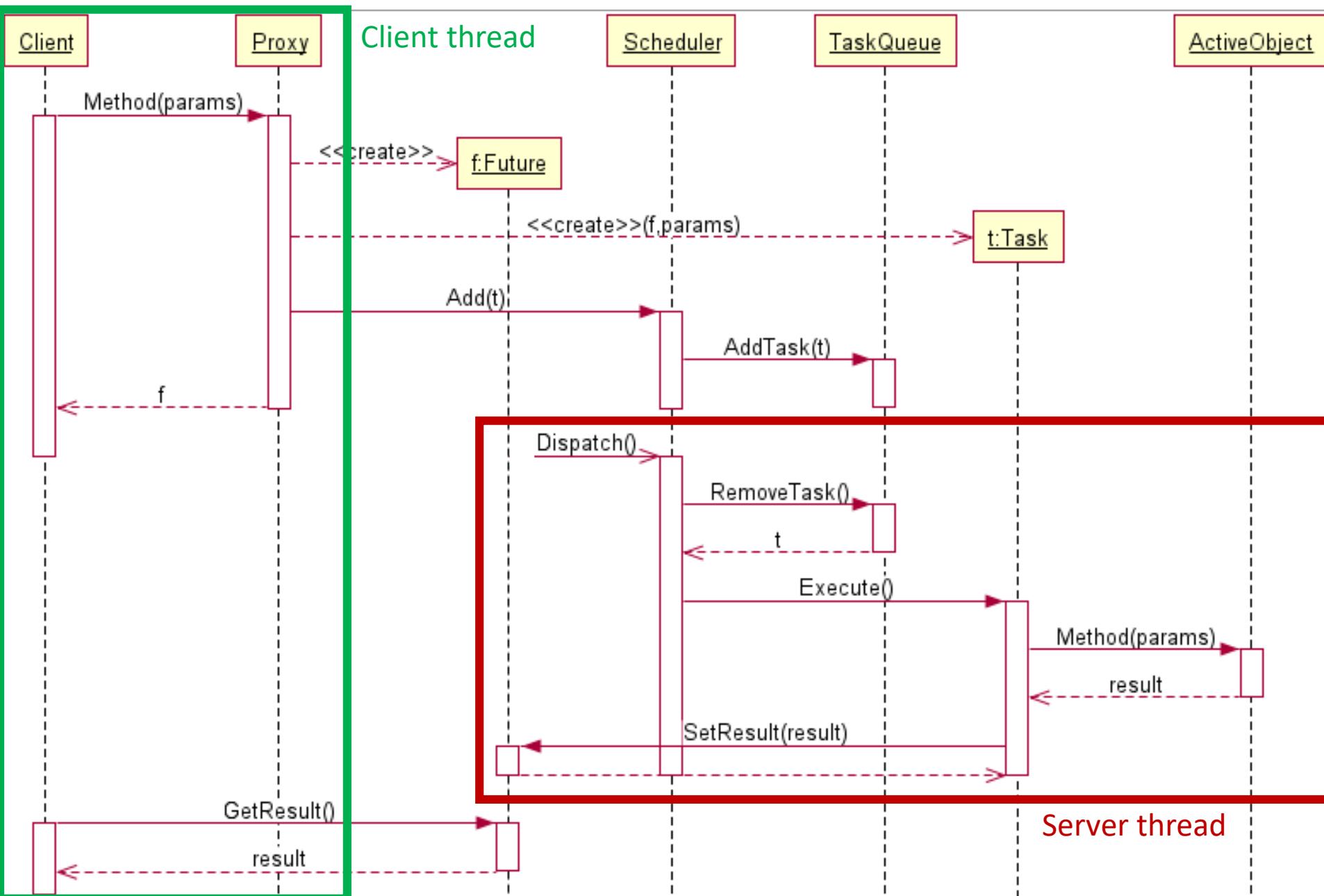
Active object

- Decouples method execution from method invocation
- Client and server reside in their own thread of control
- The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests
- The pattern consists of the following elements:
 - **Interface:** defines the methods of an active object
 - **Proxy:** implements the active object's interface, provides an interface towards clients with publicly accessible methods
 - **Task:** represents a method request
 - **Task queue:** stores pending requests from clients
 - **Scheduler:** decides which request to execute next
 - **Active object:** implementation of the active object's interface
 - **Future:** callback or variable for the client to receive the result

Active object



Active object



Active object

- The active object pattern is similar to remote communication
 - but its goal is local multithreaded implementation
- Number of active objects:
 - one per request: thread per request
 - one per client: thread per session
 - pool of active objects: thread-pool

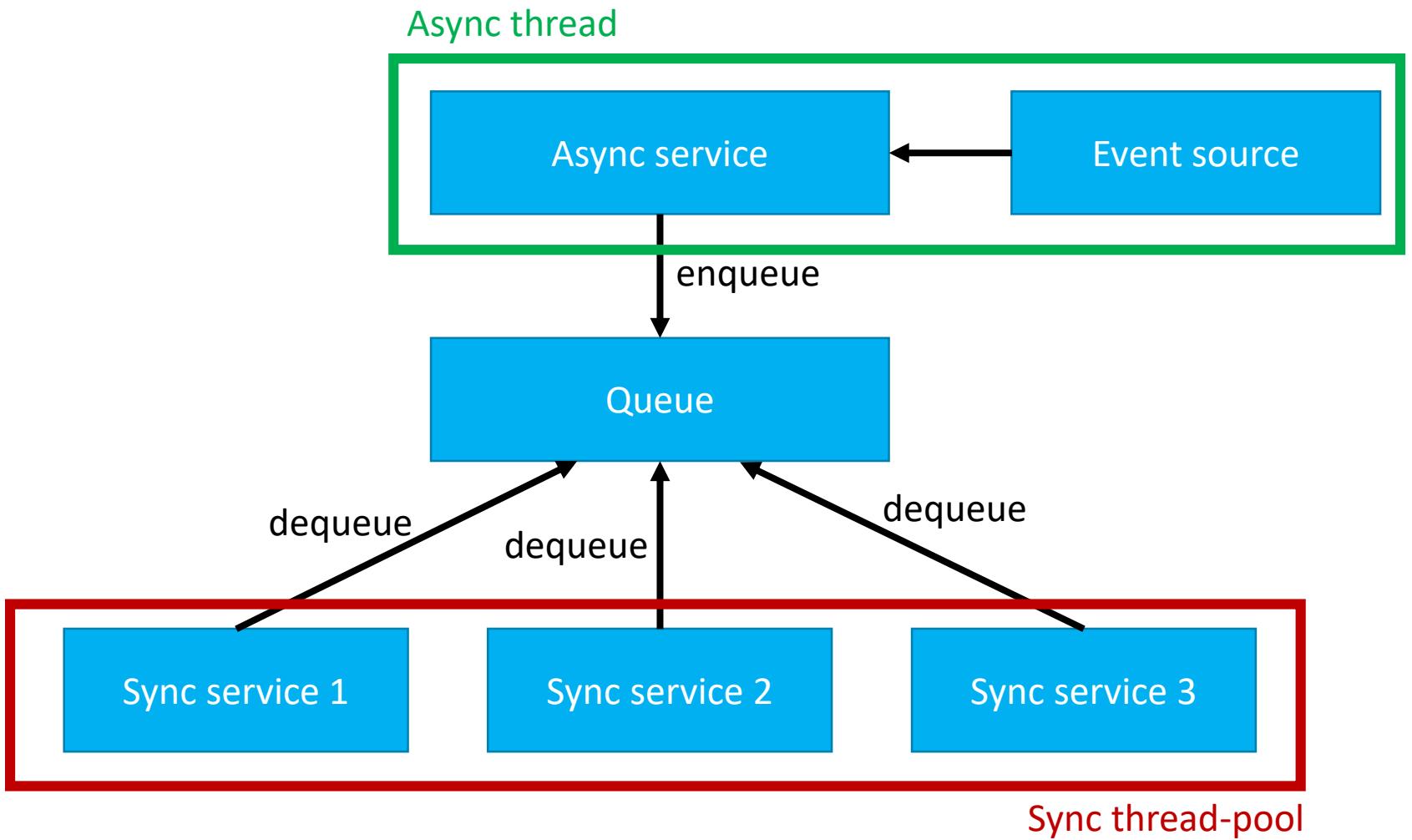
Active object examples

- .NET Tasks
- Java ExecutorService

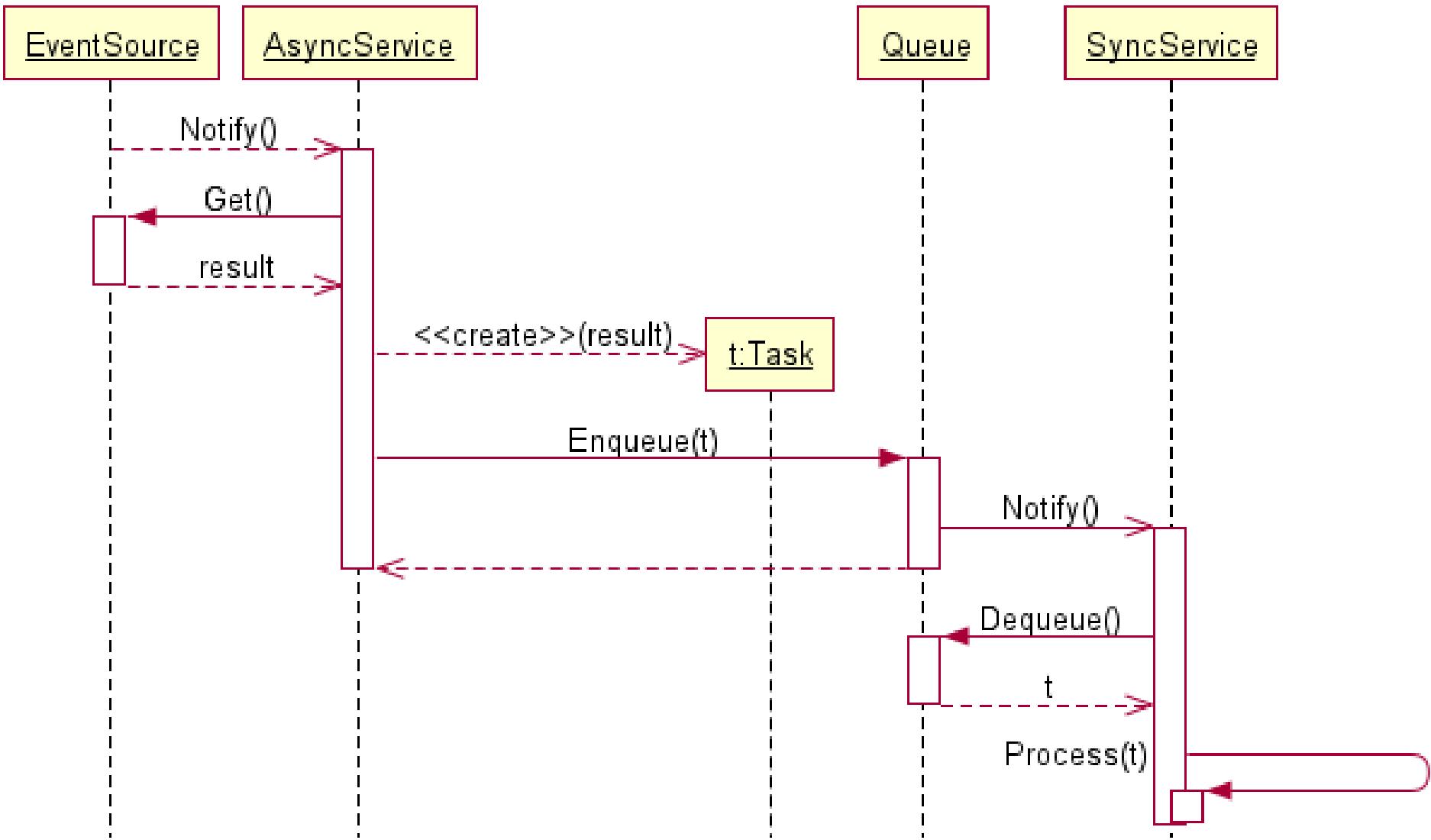
Half-sync/half-async

- Decouples asynchronous and synchronous service processing in concurrent systems, to simplify programming without unduly reducing performance
- Between the async and sync layers: queuing layer
- Two directions:
 - async -> sync
 - sync -> async
- The pattern consists of the following elements:
 - **External event source:** generates events
 - **Async service:** processes events and creates tasks from them
 - **Queue:** stores tasks to be processed
 - **Sync service:** takes a task from the queue and processes it synchronously

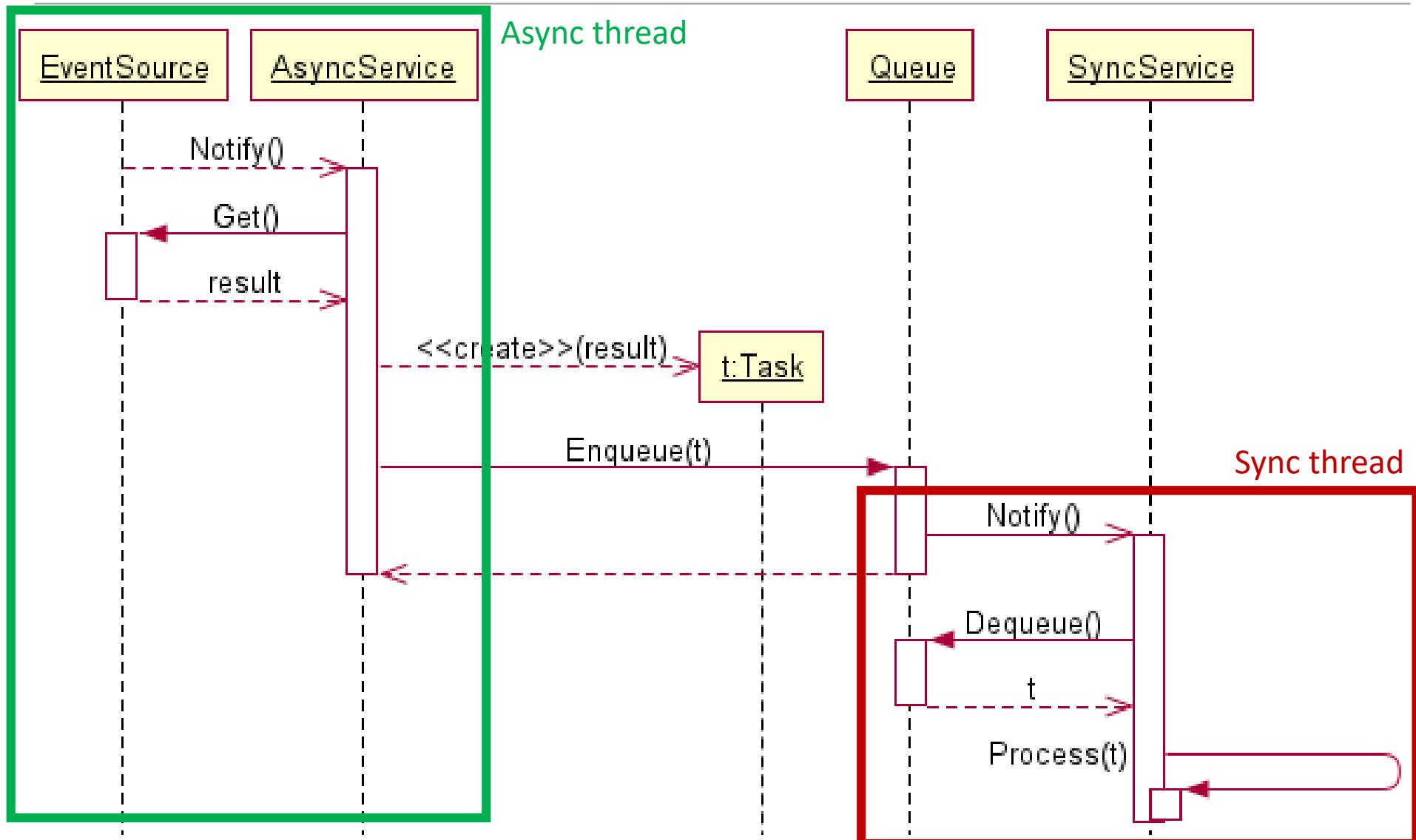
Half-sync/half-async: async -> sync



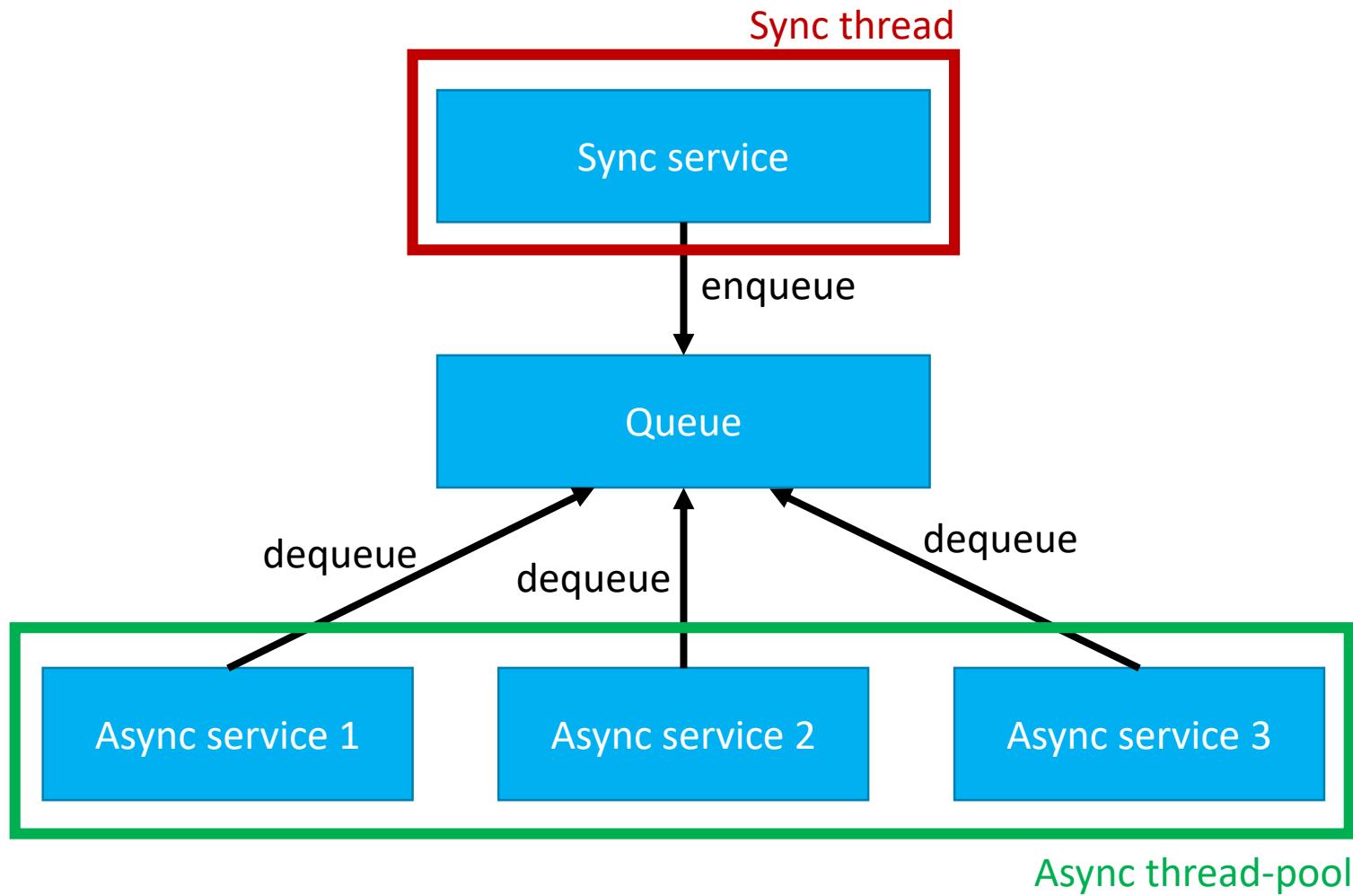
Half-sync/half-async: async -> sync



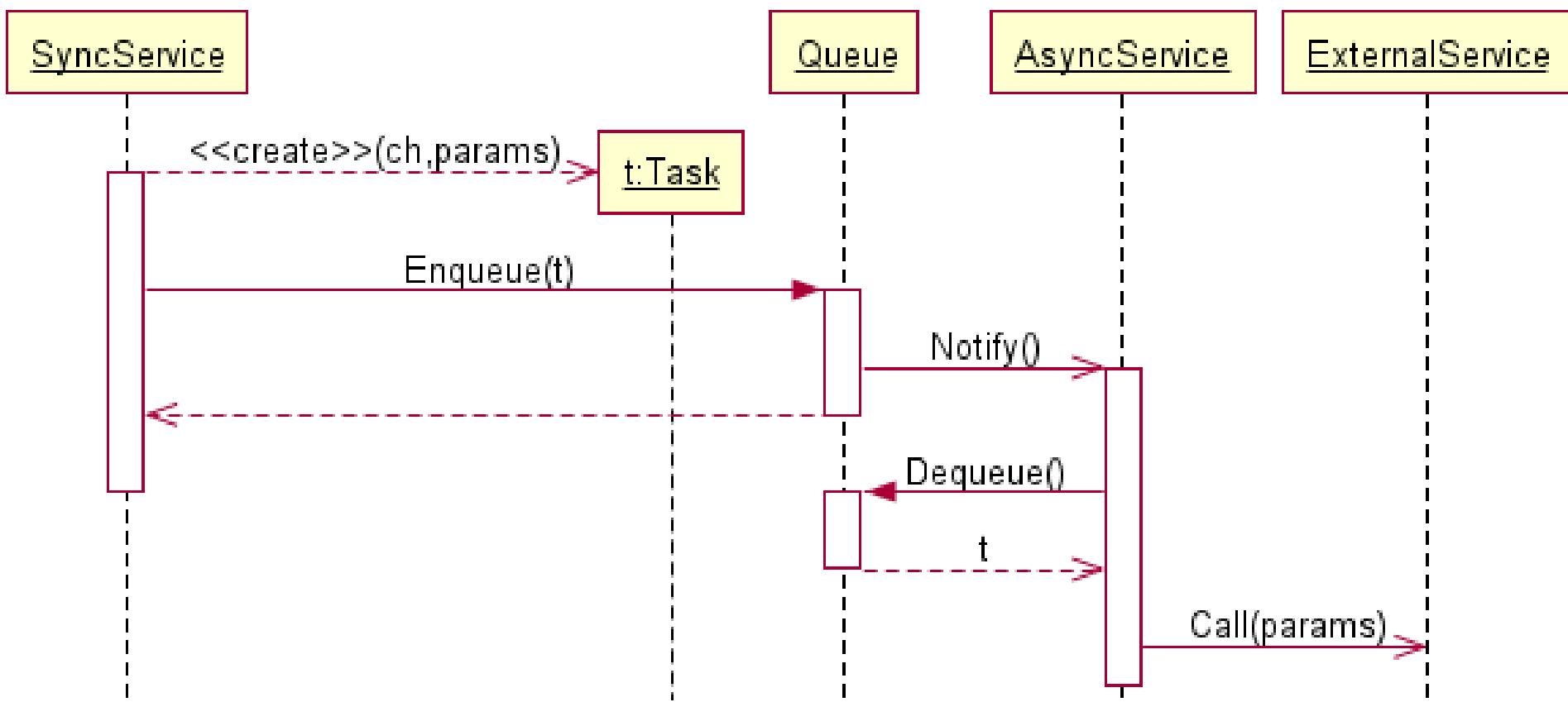
Half-sync/half-async: async -> sync



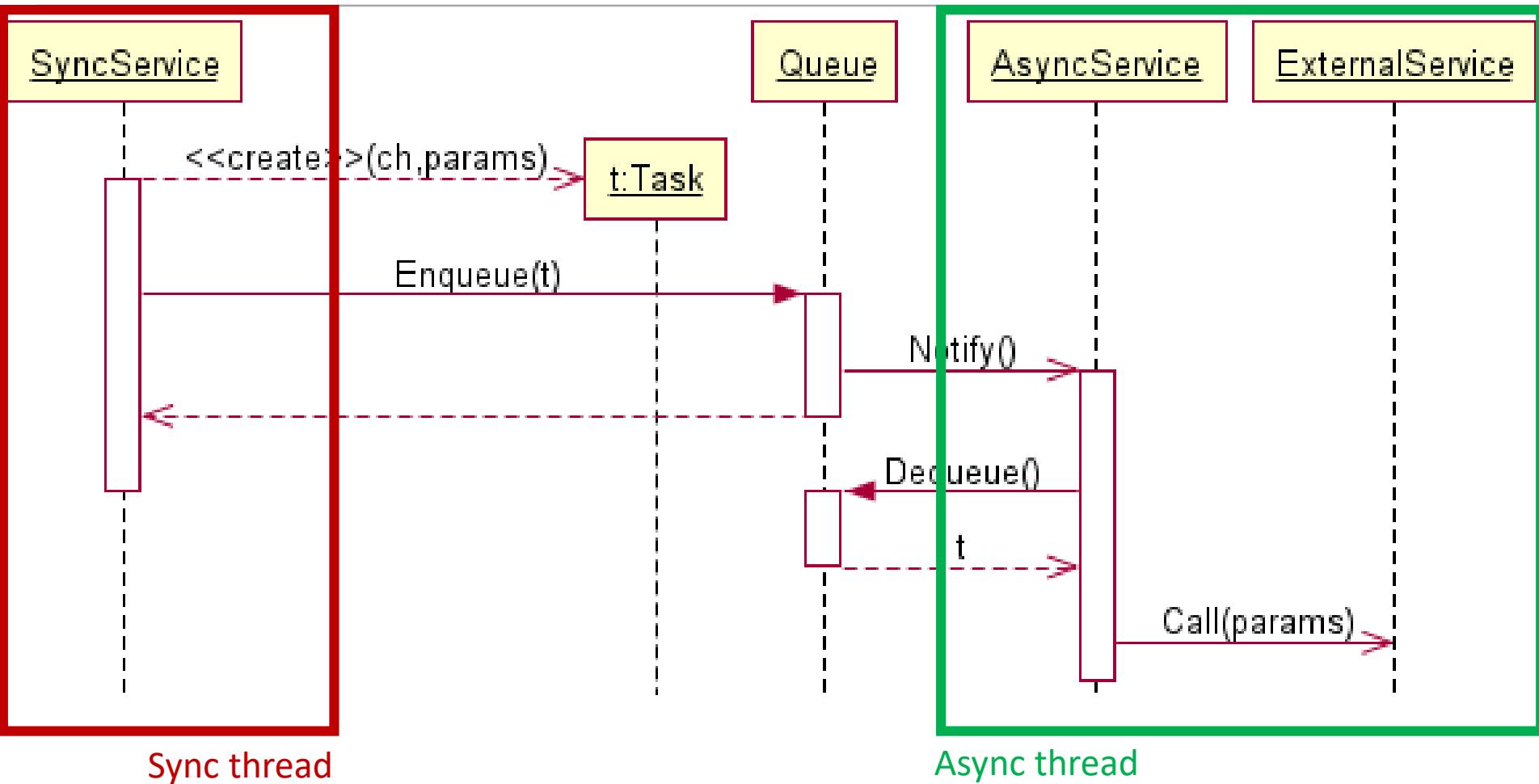
Half-sync/half-async: sync -> async



Half-sync/half-async: sync -> async



Half-sync/half-async: sync -> async



Half-sync/half-async

- The async layer could be implemented either as a Reactor or as a Proactor depending on the interface of the External event source
 - Reactor: e.g. HW interrupts
 - Proactor: e.g. async IO functions
- There can be multiple async services running on multiple threads
- If there is only one single sync thread, half-sync/half-async implements either the Reactor or the Proactor pattern:
 - Reactor: external async -> sync
 - Proactor: sync -> external async

Half-sync/half-async examples

- OS:
 - interrupts: async service
 - application: sync service
- Android
 - UI thread: async service
 - worker threads: sync service
- Reactor pattern:
 - external events: async service
 - event handlers: sync service (single thread)
- Proactor pattern:
 - initiator + completion handlers: sync service (single thread)
 - external processing: async service

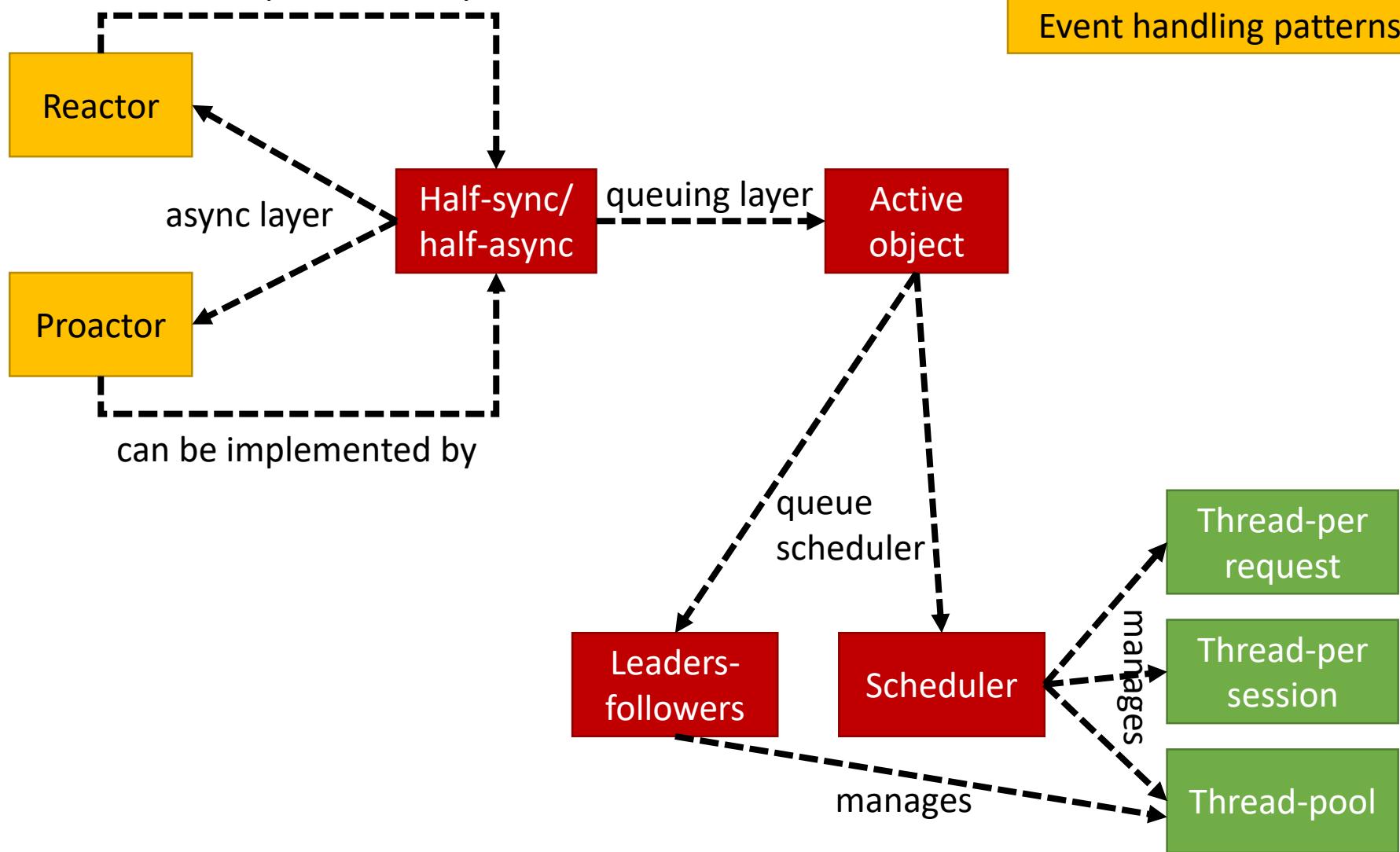
Leader-followers

- Multiple threads take turns sharing a set of event sources in order to detect, demultiplex, dispatch, and process service requests that occur on the event sources
- Two kinds of threads:
 - **Followers:** threads waiting to be leaders
 - **Leader:**
 - this thread takes the next event (task) from the queue
 - promotes one of the other follower threads to be the new leader
 - processes the event (task)
 - finally it becomes another follower waiting to be promoted

Scheduler

- A more advanced way to detect, demultiplex, dispatch, and process service requests that occur on event sources
- The scheduler manages a number of threads
 - e.g. thread-pool
- The scheduler takes events (tasks) from the event queue and assigns them to a free thread
- The scheduler can reorder events in the queue based on various conditions and on the events' priorities
- A scheduler has more control than the leader-followers pattern, but it is usually more complicated

Connection between the concurrency and event handling patterns



Immutability

Objektumorientált szoftvertervezés

Object-oriented software design

Dr. Balázs Simon

BME, IIT

Outline

- Immutability
- Problems with mutability
- Advantages of immutability
- Immutable Object-Orientation
- Disadvantages of immutability
- Immutable collections

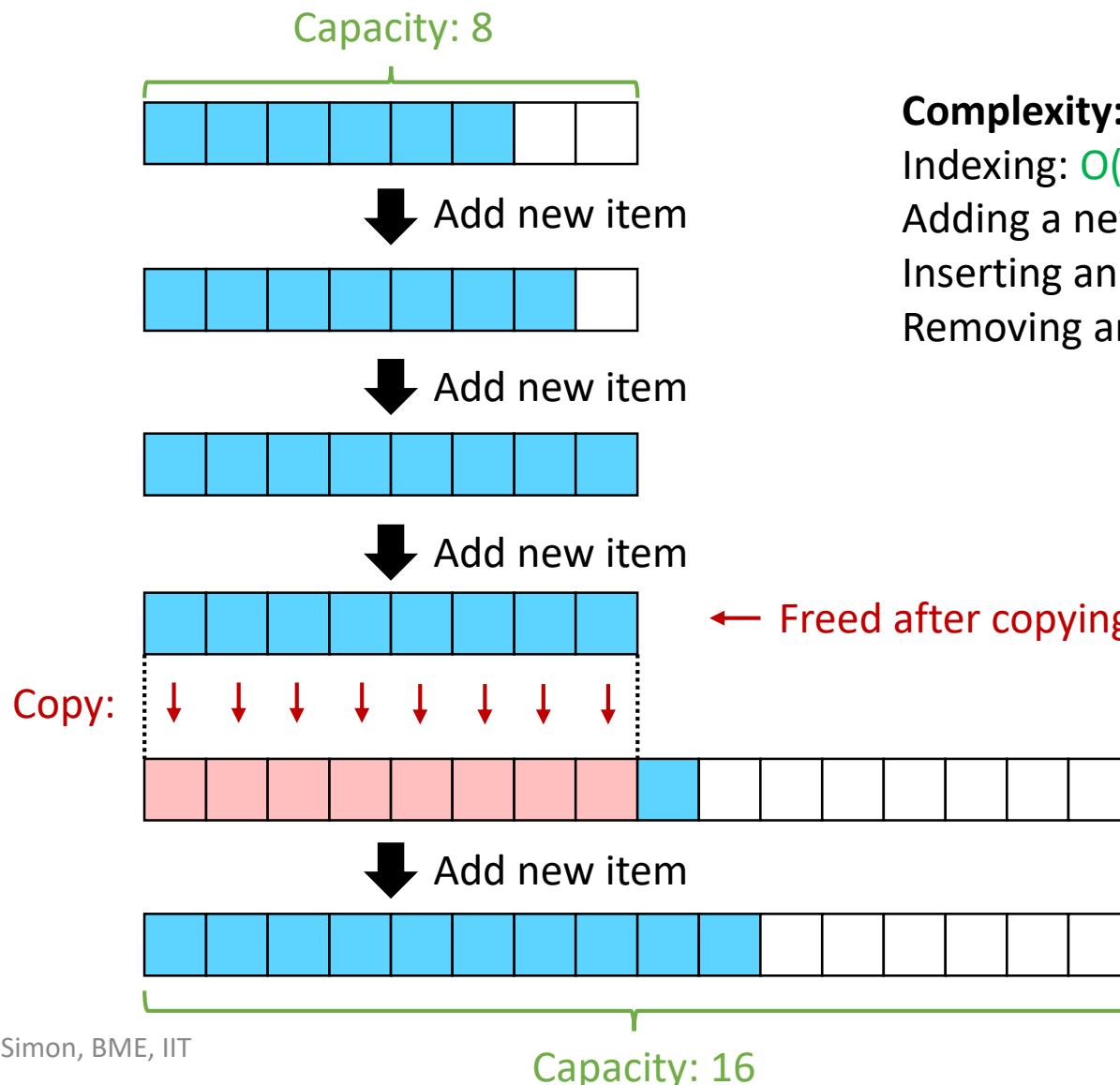
Immutability

Immutability

- An object is immutable if its state cannot change after it is constructed
 - initialized in the constructor
 - no setters
 - no methods that can change state
- Example: String class in Java and in .NET
- Greatest advantage of immutability: immutable classes are inherently thread-safe
 - no need for all those complicated thread-safety patterns
- In imperative languages we are used to mutable objects. So how can we change the state of an immutable object?
 - we can't
 - we have to create a completely new object
- Question: isn't this copying slow and wasteful?
 - Not necessarily, the unchanged parts can be reused!

Mutable list: List<T>

Mutable list is usually implemented using an array:



Complexity:

Indexing: $O(1)$

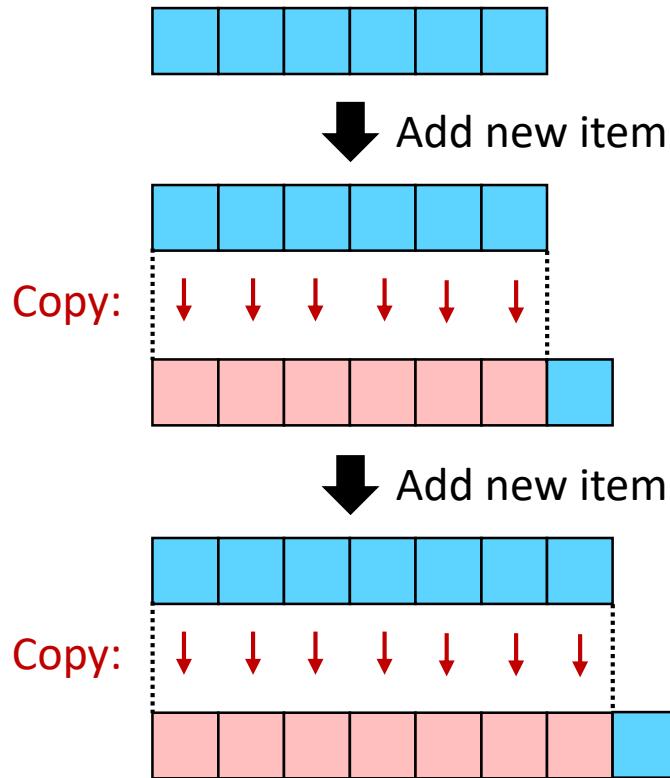
Adding a new item: $O(1)$ *on average*

Inserting an item: $O(n)$

Removing an item: $O(n)$

Immutable list with an array

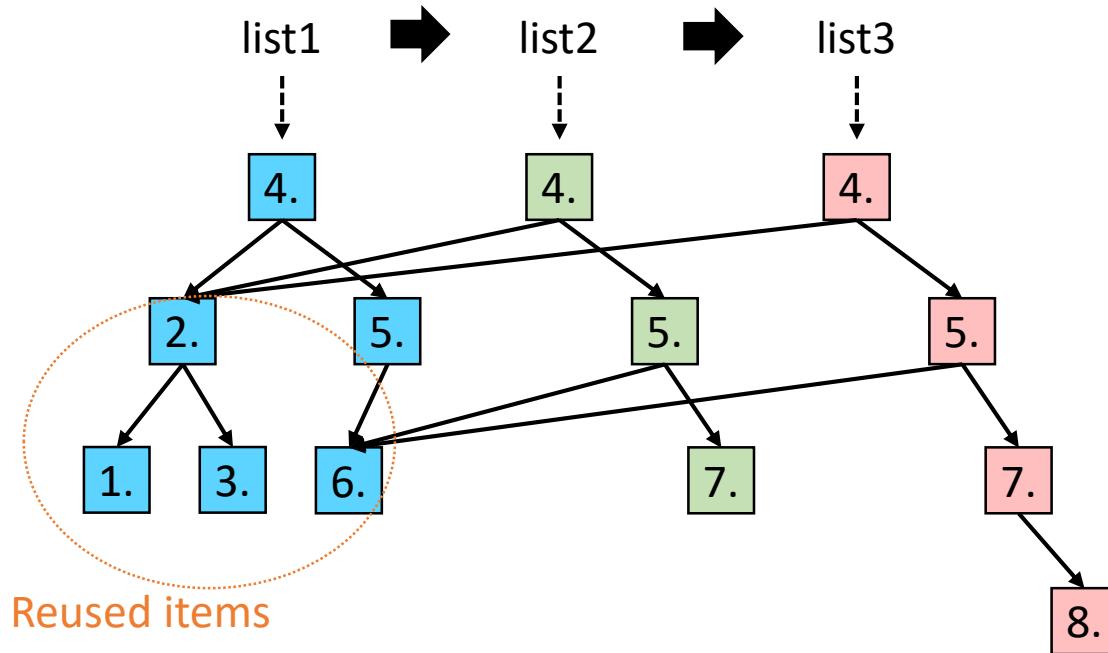
Implementing an immutable list using an array would be inefficient:
capacity is not useful, the array always has to be copied



Complexity:
Indexing: $O(1)$
Adding a new item: $O(n)$
Inserting an item: $O(n)$
Removing an item: $O(n)$

Immutable list: ImmutableList<T>

Implementing an immutable list using immutable balanced binary trees:



```
list2 = list1.Add(item)  
list3 = list2.Add(item)
```

If any one of the lists are released
the difference is collected by the GC!
No memory leaks!

Problems with mutability

Problems with mutability

- readonly/final
- Passing mutable values
- Returning mutable values
- Returning defensive copies
- Multi-threaded access
- Mutable identity
- Corrupted internal state on failure
- Temporal coupling

readonly/final

- A readonly/final field in C#/Java does not mean that the stored object is constant (immutable)
- Only that the reference stored in the field cannot be changed
- The target object of the reference is still mutable:

```
public class Date
{
    public int Year { get; set; }
    public int Month { get; set; }
    public int Day { get; set; }
}

public class Program
{
    private static readonly Date Zero = new Date();

    public static void Main(string[] args)
    {
        Zero.Year = 2000; // Zero is still mutable
    }
}
```

- (In C++ a const field is really immutable, unless you use const_cast...)

Passing mutable values

- A function may change a mutable parameter value:

```
public int Min(List<int> values)
{
    values.Sort();
    return values.First();
}
```

- The caller may not expect the change

Returning mutable values

- If an internal representation is returned:

```
public class String
{
    private char[] chars;
    // ...
    public char[] GetChars()
    {
        return this.chars;
    }
}

public class Program
{
    public static void Main(string[] args)
    {
        String str = new String("Hello");
        char[] chars = str.GetChars();
        chars[0] = 'B';
    }
}
```

- The caller may change the returned value

Returning defensive copies

- To prevent callers changing an internal representation, a defensive copy has to be returned:

```
public class String
{
    private char[] chars;

    // ...

    public char[] GetChars()
    {
        return (char[])this.chars.Clone();
    }
}
```

- This is inefficient, since every call on GetChars() makes a copy
- But at least we are protected from callers

Multi-threaded access

- Multiple threads calling the same object may leave it in an inconsistent state
 - two threads entering Push at the same time may end up overwriting each other's items
 - two threads entering Pop at the same time may see the same item popped
- Locking is required to access the Stack class from multiple threads
 - and all the complicated threading patterns are needed...

```
public class Stack<T>
{
    private int top = 0;
    private T[] items = new T[100];

    public void Push(T item)
    {
        items[top++] = item;
    }
    public T Pop()
    {
        return items[top--];
    }
}
```

Mutable identity

- The state, and therefore the identity of an object can change
- The hash-code of a mutable object can also change
- This is a problem especially if the object is used as a key in a Dictionary/HashMap

```
Dictionary<Person, string> map = new Dictionary<Person, string>();
```

```
Person p = new Person("Alice");  
map.Add(p, "Hello");
```

```
p.Name = "Bob";  
string value = map[p]; // KeyNotFoundException
```

Corrupted internal state on failure

- A mutable object can be left in an inconsistent state when an error occurs:

```
public class Stack {  
    private int size;  
    private String[] items;  
    // ...  
    public void push(String item) {  
        size++;  
        if (size > items.length) {  
            throw new RuntimeException("stack overflow");  
        }  
        items[size-1] = item;  
    }  
}
```

- If an exception is thrown, the value of “size” is inconsistent with the “items” array

Temporal coupling

- Example:

```
Request request = new Request("http://example.com");
request.method("POST");
String first = request.fetch();
request.body("text=hello"); // modifies the request object
String second = request.fetch();
```

- Problem: we reuse the configuration of “first” in the configuration of “second”

- they are in temporal coupling
- if we reconfigure “first”, we also reconfigure “second”
- they have to stay together in this order
- this is a hidden information in code that we have to remember

Advantages of immutability

Advantages of immutable objects

- Simple to construct, test, and use
- Always thread-safe
- Don't need copy constructor and cloning
- Side-effect free
- No identity mutability
- Failure atomicity
- Easier to cache
- Prevent NULL references, which are bad
- Avoid temporal coupling

Simple to construct, test, and use

- Constructing immutable objects:
 - initialized in the constructor
 - setters do not modify, they return a different immutable object
- Examples:

```
ImmutablePerson p =  
    new ImmutablePerson(firstName: "Bob", lastName: "White", age: 45);
```

```
ImmutableList<string> l = ImmutableList<string>.Empty;  
ImmutableList<string> l2 = l.Add("Bob");
```

Always thread-safe

- Immutable objects can only be read
- No matter how many of them and how often are being called parallel, they are always thread-safe
- Multiple threads can access the same object at the same time, without clashing with another thread

Don't need copy constructor and cloning

- Never need to make a copy of an immutable object
 - since nobody will be able to change it
- It is enough to keep the reference
 - much more efficient than copying the whole object
- A single immutable instance can be used in many places (e.g. default values like empty lists)
 - uses less memory
 - faster: no new instances needed
- Example:

```
public ImmutableList<Person> FindPersons(string name)
{
    if (name == null) return ImmutableList<Person>.Empty;
    // ...
}
```

Side-effect free

- Can be freely passed as parameters:

```
public int Min(ImmutableList<int> values)
{
    ImmutableList<int> sorted = values.Sort(); // "values" is unchanged
    return sorted.First();
}
```

- Internal representations can be freely returned:

```
public class String
{
    private ImmutableList<char> chars = ImmutableList<char>.Empty;
    // ...
    public ImmutableList<char> GetChars()
    {
        return this.chars; // caller won't be able to modify this
    }
}
```

- No defensive copies are needed

No identity mutability

- An immutable object's identity is its own state
- The state never changes, therefore, the identity cannot change
 - hence, its `HashCode` cannot change

```
ImmutablePerson p = new ImmutablePerson("Alice");
map.Add(p, "Hello");
```

```
ImmutablePerson p2 = p.SetName("Bob");
```

```
string value = map[p]; // OK, "p" cannot be changed
string value2 = map[p2]; // KeyNotFoundException
```

```
ImmutablePerson p3 = new ImmutablePerson("Alice");
string value3 = map[p3]; // OK, "p3" has the same identity as "p"
```

Failure atomicity

- An immutable object will never be left in a broken state
- Its state is modified only in its constructor
 - the constructor either succeeds and the object is in a consistent state
 - or the constructor fails and the object is not created

```
public class ImmutableListStack {  
    private final int size;  
    private final ImmutableList<String> items;  
    // ...  
    private ImmutableListStack(int size, ImmutableList<String> items) {  
        this.size = size;  
        this.items = items;  
    }  
    public ImmutableListStack push(String item) {  
        if (size+1 > items.length) {  
            throw new RuntimeException("stack overflow");  
        }  
        return new ImmutableListStack(size+1, items.set(size, item));  
    }  
}
```

Easier to cache

- If all the parts of two immutable objects are the same, the two immutable objects are also the same

```
public AddOrSubExpression CreateAddOrSubExpression(Expression left,  
                                                 Token _operator, Expression right)  
{  
    int hash;  
    var cached =  
        SyntaxNodeCache.TryGetNode(SyntaxKind.AddOrSubExpression,  
                                    left, _operator, right, out hash);  
    if (cached != null) return (AddOrSubExpression)cached;  
    var result =  
        new AddOrSubExpressionGreen(SyntaxKind.AddOrSubExpression,  
                                    left, _operator, right);  
    if (hash >= 0)  
    {  
        SyntaxNodeCache.AddNode(result, hash);  
    }  
    return result;  
}
```

Prevent NULL references, which are bad

- Clean-code:
 - don't pass null
 - don't return null
- These rules are easy to keep with immutable objects
 - pass or return: default object instead of null
 - e.g. empty collections
 - efficient:
 - no need to create new default objects, new empty lists, etc. on the fly
 - the same immutable default object instance can be used everywhere

```
ImmutableList<int> list = ImmutableList<int>.Empty;
if (!error)
{
    list = list.AddRange(items);
}
ImmutableList<int> sortedList = Sort(list);
```

Avoid temporal coupling

- Example:

```
final Request request = new Request("");
final Request post = request.method("POST");
String first = post.fetch();
String second = post.body("text=hello").fetch();
```

- No temporal coupling:

- “first” and “second” are independent of each other
- both of them are derived from the same “post” object

Immutable Object-Orientation

Example: mutable Date

```
public class MutableDate
{
    private int year;
    private int month;
    private int day;

    public MutableDate(int year = 0, int month = 0, int day = 0)
    {
        this.year = year;
        this.month = month;
        this.day = day;
    }
    public int Year
    {
        get { return this.year; }
        set { this.year = value; }
    }
    public int Month
    {
        get { return this.month; }
        set { this.month = value; }
    }
    public int Day
    {
        get { return this.day; }
        set { this.day = value; }
    }
}
```

Date

-year: int
-month: int
-day: int

Immutable Object-Orientation

- Design pattern:
 - immutable class
 - initialized in the constructor
 - getters for the fields (Get...)
 - methods for incremental change as a fluent API (With...)
 - a method for multiple changes (Update)
 - other methods for behavior
 - a method for creating a builder from the current object (ToBuilder)
 - a static method for creating a builder (CreateBuilder)
 - builder class
 - mutable: not thread safe!
 - the builder can be more efficient when there are a lot of changes
 - e.g. adding items to a list
 - getters for the fields (Get...)
 - setters for the fields (Set...)
 - setters for the fields as a fluent API (With...)
 - a method for returning the immutable object from the current state (ToImmutable)

Example: immutable Date (1/5)

```
public class Date
{
    // Immutable default value:
    public static readonly Date Default = new Date();

    // Fields are readonly to prevent accidental change:
    private readonly int year;
    private readonly int month;
    private readonly int day;

    public Date(int year = 0, int month = 0, int day = 0)
    {
        this.year = year;
        this.month = month;
        this.day = day;
    }

    // Getters:
    public int Year { get { return this.year; } }
    public int Month { get { return this.month; } }
    public int Day { get { return this.day; } }

    // ... next slide ...
}
```

Date

-year: int
-month: int
-day: int

Example: immutable Date (2/5)

```
public class Date
{
    // ... previous slide ...

    // Update multiple fields:
    public Date Update(int year, int month, int day)
    {
        if (year != this.year || month != this.month || day != this.day)
        {
            return new Date(year, month, day);
        }
        return this;
    }

    // Update individual fields as a fluent API:
    public Date WithYear(int year)
    {
        return this.Update(year, this.month, this.day);
    }
    public Date WithMonth(int month)
    {
        return this.Update(this.year, month, this.day);
    }
    public Date WithDay(int day)
    {
        return this.Update(this.year, this.month, day);
    }
    // ... next slide ...
}
```

Date

-year: int
-month: int
-day: int

Example: immutable Date (3/5)

```
public class Date {  
    // ... previous slide ...  
  
    // Builder from the current object:  
    public Date.Builder ToBuilder()  
    {  
        return new Date.Builder(this);  
    }  
    // Builder from the default value:  
    public static Date.Builder CreateBuilder()  
    {  
        return new Date.Builder(Date.Default);  
    }  
  
    // Inner class of Date:  
    public class Builder  
    {  
        private int year;  
        private int month;  
        private int day;  
  
        internal Builder(Date date)  
        {  
            this.year = date.Year;  
            this.month = date.Month;  
            this.day = date.Day;  
        }  
        // ... next slide ...  
    }  
}
```

Date

-year: int
-month: int
-day: int

Example: immutable Date (4/5)

```
public class Date
{
    public class Builder
    {
        // ... previous slide ...

        // Getters-setters:
        public int Year
        {
            get { return this.year; }
            set { this.year = value; }
        }

        public int Month
        {
            get { return this.month; }
            set { this.month = value; }
        }

        public int Day
        {
            get { return this.day; }
            set { this.day = value; }
        }

        // ... next slide ...
    }
}
```

Date

-year: int
-month: int
-day: int

Example: immutable Date (5/5)

```
public class Date {  
    public class Builder {  
        // ... previous slide ...  
  
        // Setters for the fluent API:  
        public Builder WithYear(int year)  
        {  
            this.Year = year;  
            return this;  
        }  
        public Builder WithMonth(int month)  
        {  
            this.Month = month;  
            return this;  
        }  
        public Builder WithDay(int day)  
        {  
            this.Day = day;  
            return this;  
        }  
  
        // Construct an immutable object from the current state of the builder:  
        public Date ToImmutable()  
        {  
            return new Date(this.year, this.month, this.day);  
        }  
    }  
}
```

Date

-year: int
-month: int
-day: int

Using the immutable Date

```
Date d1 = Date.Default; // 0.0.0.  
Date d2 = d1.WithYear(2016); // 2016.0.0.  
Date d3 = d1.WithYear(2017).WithMonth(9); // 2017.9.0.  
Date d4 = d3.Update(d3.Year, 5, 23); // 2017.5.23.  
  
Date.Builder b1 = Date.CreateBuilder(); // 0.0.0.  
b1.Year = 2016;  
b1.Month = 9;  
Date d5 = b1.ToImmutable(); // 2016.9.0.  
b1.Day = 27;  
Date d6 = b1.ToImmutable(); // 2016.9.27.  
  
Date.Builder b2 = d4.ToBuilder(); // 2017.5.23.  
b2.Month = 3;  
b2.Day = 21;  
Date d7 = b2.ToImmutable(); // 2017.3.21.  
  
// 2017.6.18.  
Date d8 = d7.ToBuilder().WithMonth(6).WithDay(18).ToImmutable();
```

Example: immutable Person (1/2)

```
public class Person
{
    public static readonly Person Default = new Person();

    private readonly string name;
    private readonly double height;
    private readonly Date birthDate;

    public Person()
        : this(string.Empty, 0, Date.Default)
    {
    }

    public Person(string name, double height, Date birthDate)
    {
        this.name = name;
        this.height = height;
        this.birthDate = birthDate;
    }

    public string Name { get { return this.name; } }
    public double Height { get { return this.height; } }
    public Date BirthDate { get { return this.birthDate; } }

    // ...
}
```

Person

-name: String
-height: double
-birthDate: Date

Example: immutable Person (2/2)

```
public class Person
{
    // ...
    public class Builder
    {
        private string name;
        private double height;
        private Date.Builder birthDate; // Date builder!

        internal Builder(Person person)
        {
            this.name = person.Name;
            this.height = person.Height;
            // Immutable date to date builder:
            this.birthDate = person.BirthDate.ToBuilder();
        }

        public Person ToImmutable()
        {
            // Date builder to immutable date:
            return new Person(this.name, this.height, this.birthDate.ToImmutable());
        }
    }
}
```

Person

-name: String
-height: double
-birthDate: Date

Converting between immutable objects and their builders can be quite expensive!
Only worth's it if a lot of operations have to be performed
and the builder is more efficient for those operations.
Sometimes it's just not worth it to have a builder at all.

Disadvantages of immutability

Disadvantages of immutability

- Requires a lot of plumbing
- Inconvenient syntax
- Cheaper to update an existing object than to create a new one
- A small change in a large immutable structure is very inconvenient
- Conversion between immutable objects and builders is inefficient
- No circular reference possible

Requires a lot of plumbing

- Mutable objects: fields + getter-setters
 - mutable date: ~25 lines
- Immutability:
 - Immutable objects: fields + getters + with methods + to builder
 - immutable date: ~50 lines
 - Builder objects: fields + getter-setters + with methods + to immutable
 - immutable date builder: ~50 lines
- ~4x the code!
 - mechanical work
 - but can be generated automatically

Inconvenient syntax

- Imperative languages are designed for mutability
- Immutable objects in imperative languages have an inconvenient syntax

Mutable is more intuitive:

```
Date mutableDate = new Date();
mutableDate.Year = 2017;
mutableDate.Month = 9;
```

Immutable is a bit inconvenient (although still readable):

```
Date immutableDate = Date.Default.WithYear(2017).WithMonth(9);
```

Cheaper to update an existing object than to create a new one

- Imperative languages are designed for mutability
 - because mutability is very efficient
- Modifying a mutable field frequently is more efficient than creating a new object every time
 - especially in large structures
- Examples:
 - a Word document as an immutable object
 - every time you type a character, the whole document must be recreated (although most of it can be reused)
 - a Visual Studio solution with projects and source files
 - every time you type a character, the whole solution must be recreated (although most of it can be reused)
 - Roslyn actually does this!

A small change in a large immutable structure is very inconvenient

- If there is a change in an immutable structure, the whole structure must be recreated
 - no matter how small the change is
- Although most of the structure can be reused
- The recreation requires a lot of (recursive) calls
 - see functional languages where everything is immutable
 - a lot of functions need to be created for a small change deep in the structure
- However, if the structure is traversed as a whole and the recreation functions are there, this is not a problem
 - e.g. immutable binary trees, Roslyn syntax trees, etc.

Conversion between immutable objects and builders is inefficient

- Builders are more efficient if changes are frequent
 - since builders are mutable
 - (but they are usually not thread-safe: cannot be used from multiple threads)
- Converting an immutable object structure to a builder object structure or a builder object structure to an immutable object structure is inefficient
 - requires a lot of copying
 - only worth's it, if there are a lot of changes performed on the builders
 - otherwise, it is more efficient to just omit the builders and use the With... functions of the immutable objects
- Sometimes builders can use the same underlying data structure as the immutable object

No circular reference possible

- Immutable data structures must be DAGs
 - usually they are trees
- No circular references are possible
 - an immutable object cannot be changed after it is created:

```
Husband h = new Husband(???);  
Wife w = new Wife(h);
```

- Possible solutions:
 - bi-directional relationships can be stored as separate immutable objects
 - e.g. Marriage
 - lazy initialization
 - (but it makes things a bit more complicated)

```
public class Husband  
{  
    private Wife wife;  
  
    public Husband(Wife wife)  
    {  
        this.wife = wife;  
    }  
  
    public class Wife  
{  
        private Husband husband;  
  
        public Wife(Husband husband)  
        {  
            this.husband = husband;  
        }  
    }  
}
```

Immutable collections

Advantages of immutable collections

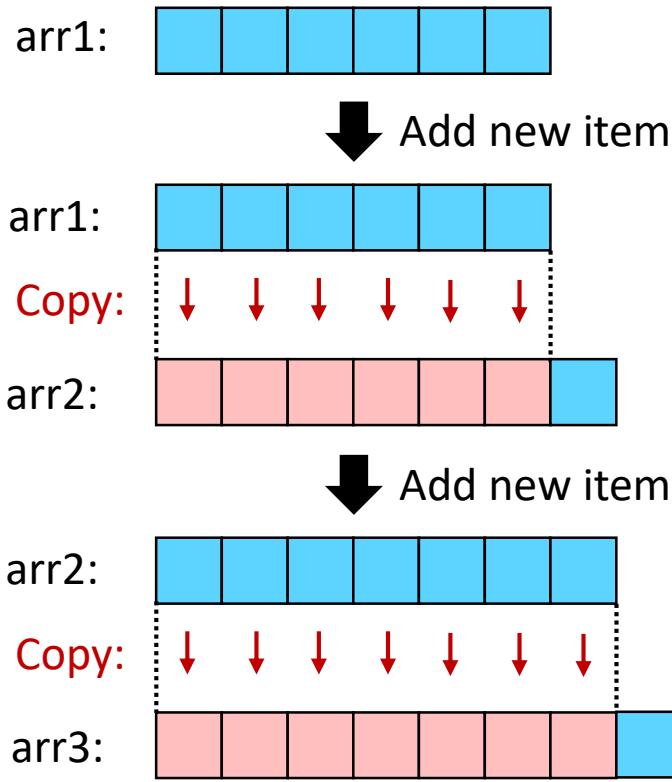
- Snapshot semantics: allowing you to share your collections in a way that the receiver can count on never changing
- Implicit thread-safety in multi-threaded applications: no locks required to access collections
- Any time you have a class member that accepts or returns a collection type and you want to include read-only semantics in the contract
- Functional programming friendly
- Allow modification of a collection during enumeration, while ensuring original collection does not change
- They implement the same `IReadOnly*` interfaces that your code already deals with so migration is easy

System.Collections.Immutable

- Structures:
 - `ImmutableArray<T>`
- Classes:
 - `ImmutableStack<T>`
 - `ImmutableQueue<T>`
 - `ImmutableList<T>`
 - `ImmutableHashSet<T>`
 - `ImmutableSortedSet<T>`
 - `ImmutableDictionary<K, V>`
 - `ImmutableSortedDictionary<K, V>`

ImmutableArray<T>

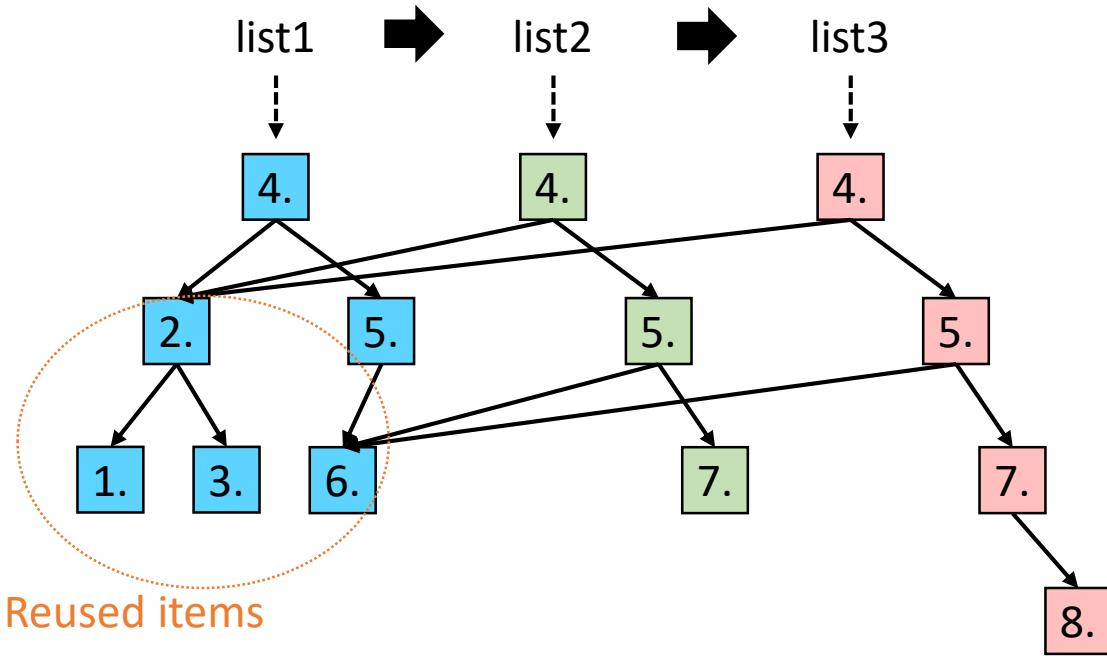
```
arr2 = arr1.Add(item)  
arr3 = arr2.Add(item)
```



Complexity:
Indexing: $O(1)$
Adding a new item: $O(n)$
Inserting an item: $O(n)$
Removing an item: $O(n)$

ImmutableList<T>

```
list2 = list1.Add(item)  
list3 = list2.Add(item)
```



Complexity:

Indexing: $O(\log n)$

Adding a new item: $O(\log n)$

Inserting an item: $O(\log n)$

Removing an item: $O(\log n)$

ImmutableArray<T> vs. ImmutableList<T>

- Reasons to use immutable array:
 - Updating the data is rare or the number of elements is quite small (less than 16 items)
 - You need to be able to iterate over the data in performance critical sections
 - You have many instances of immutable collections and you can't afford keeping the data in trees
- Reasons to use immutable list:
 - Updating the data is common or the number of elements isn't expected to be small
 - Updating the collection is more performance critical than iterating the contents

Using the immutable collections: Empty

```
var emptyFruitBasket = ImmutableList<string>.Empty;
```

No constructor: no memory allocation is needed

Empty is a static singleton: can be shared throughout the application

Using the immutable collections

```
var emptyFruitBasket = ImmutableList<string>.Empty;  
  
var fruitBasketWithApple = emptyFruitBasket.Add("Apple");
```

Each operation returns a new collection

The original emptyFruitBasket is still empty

Using the immutable collections

```
var fruitBasket = ImmutableList<string>.Empty;  
fruitBasket = fruitBasket.Add("Apple");  
fruitBasket = fruitBasket.Add("Banana");  
fruitBasket = fruitBasket.Add("Celery");
```

We can reassign the same local variable

Intermediate lists are still not modified, and internal parts of them may be reused

Intermediate lists and their non-reused internal parts will be garbage collected

Using the immutable collections

```
var list = ImmutableList<int>.Empty;  
list.Add(3); // forgot to reassign result to a local variable  
             // contents of the "list" local variable is still empty!
```

Caution: make sure to save the result of an operation!

Using the immutable collections

```
fruitBasket = fruitBasket.AddRange(new[] { "Kiwi", "Lemons", "Grapes" });
```

Add several elements at once to prevent many intermediate objects

Using the immutable collections

```
var fruitBasket = ImmutableList<string>.Empty
    .Add("Apple")
    .Add("Banana")
    .Add("Celery")
    .AddRange(new[] { "Kiwi", "Lemons", "Grapes" });
```

Instead of saving the intermediate results, we can use them as a fluent API.

Builders

- Each top-level operation results in allocating a few new nodes in that binary tree
 - increases GC pressure
 - just like concatenating strings
- Immutable collections have builders
 - just like StringBuilder for strings
- Immutable collection builder:
 - it is a *mutable* collection that uses exactly the same data structure as the immutable collection, but without the immutability requirement
 - implements the same mutable interface that we are used to
 - to restore immutability call **ToImmutable()**
 - creates a snapshot: the builder freezes its internal structure and returns it as an immutable collection

Builders

```
/// <summary>Maintains the set of customers.</summary>
private ImmutableHashSet<Customer> customers;

public ImmutableHashSet<Customer> GetCustomersInDebt()
{
    // Since most of our customers are debtors, start with
    // the customers collection itself.
    var debtorsBuilder = this.customers.ToBuilder();

    // Now remove those customers that actually have positive balances.
    foreach (var customer in this.customers)
    {
        if (customer.Balance >= 0.0)
        {
            // We don't have to reassign the result because
            // the Builder modifies the collection in-place.
            debtorsBuilder.Remove(customer);
        }
    }

    return debtorsBuilder.ToImmutable();
}
```

In this example:

- no immutable collection is modified
- no collection is copied entirely

Bulk modification without Builders

```
private ImmutableHashSet<Customer> customers;

public ImmutableHashSet<Customer> GetCustomersInDebt()
{
    return this.customers.Except(c => c.Balance >= 0.0);
}
```

As efficient as the Builder approach

However, not all bulk modifications are expressible as a single method call with a lambda:
use the Builder in those cases

Immutable vs. mutable collections

Immutable collections can be faster than mutable collections:

```
private List<int> collection;

public IReadOnlyList<int> SomeProperty
{
    get
    {
        lock (this)
        {
            return this.collection.ToList();
        }
    }
}
```

```
private ImmutableList<int> collection;

public IReadOnlyList<int> SomeProperty
{
    get { return this.collection; }
}
```

Immutable vs. mutable collections

Algorithmic complexity:

	Mutable (amortized)	Mutable (worst case)	Immutable
Stack.Push	O(1)	O(n)	O(1)
Queue.Enqueue	O(1)	O(n)	O(1)
List.Add	O(1)	O(n)	O(log n)
ImmutableArray.Add			O(n)
HashSet.Add	O(1)	O(n)	O(log n)
SortedSet.Add	O(log n)	O(n)	O(log n)
Dictionary.Add	O(1)	O(n)	O(log n)
SortedDictionary.Add	O(log n)	O(n log n)	O(log n)

Immutability vs. mutability

Immutability vs. mutability

- Use immutability whenever possible: immutability is good
- Immutability protects you from a lot of bugs and threading issues
- Immutability is great for simple objects and tree hierarchies
- There are Garbage Collectors optimized for immutability:
 - an immutable object always references older objects than itself
- Mutability is usually more efficient than immutability when there are frequent changes
- However, it is harder to write thread-safe mutable code