

Object-oriented design heuristics

Objektumorientált szoftvertervezés

Object-oriented software design

Dr. Balázs Simon

BME, IIT

Outline

- Classes
- Responsibilities
- Associations
- Inheritance

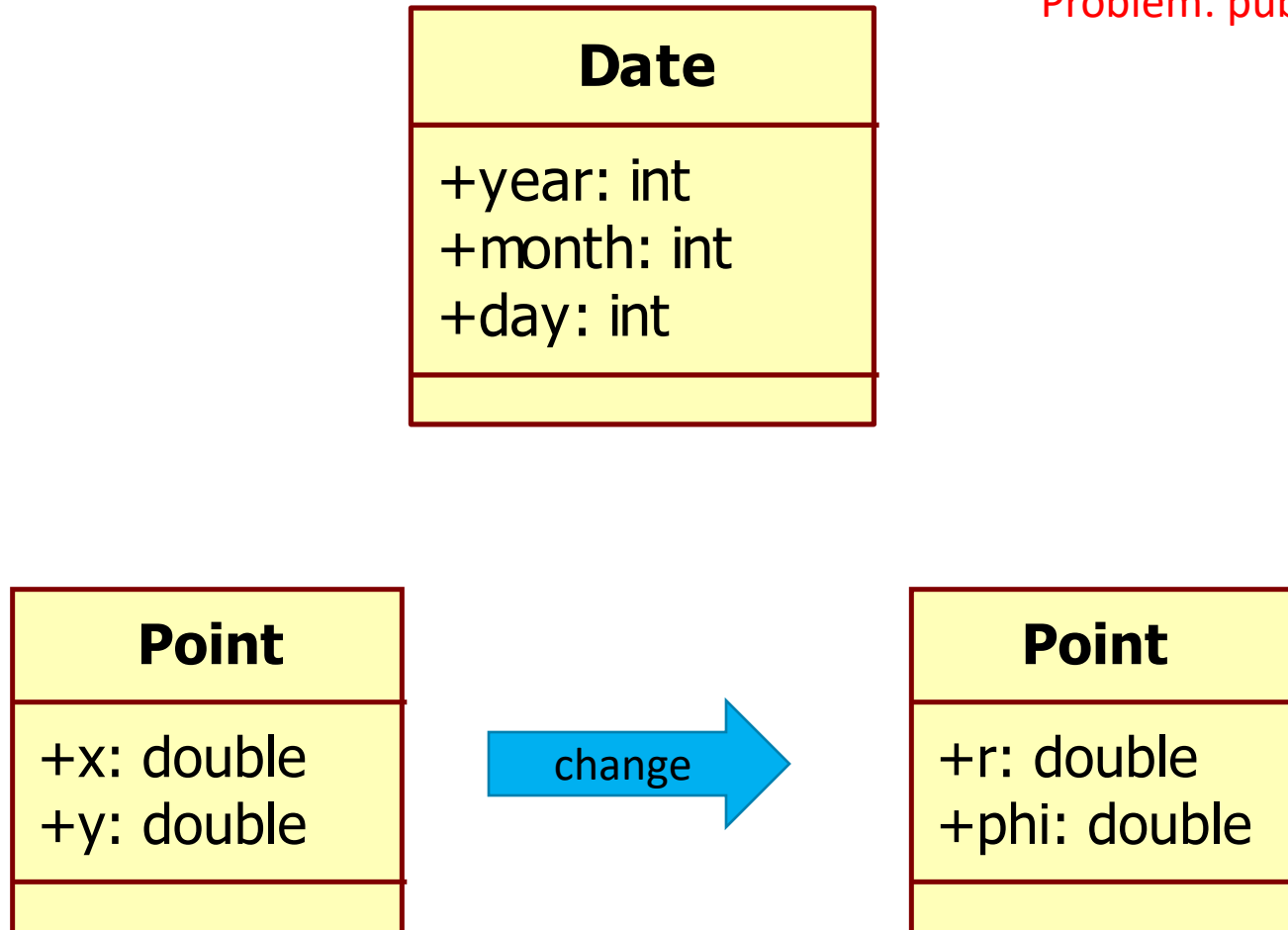
Classes

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



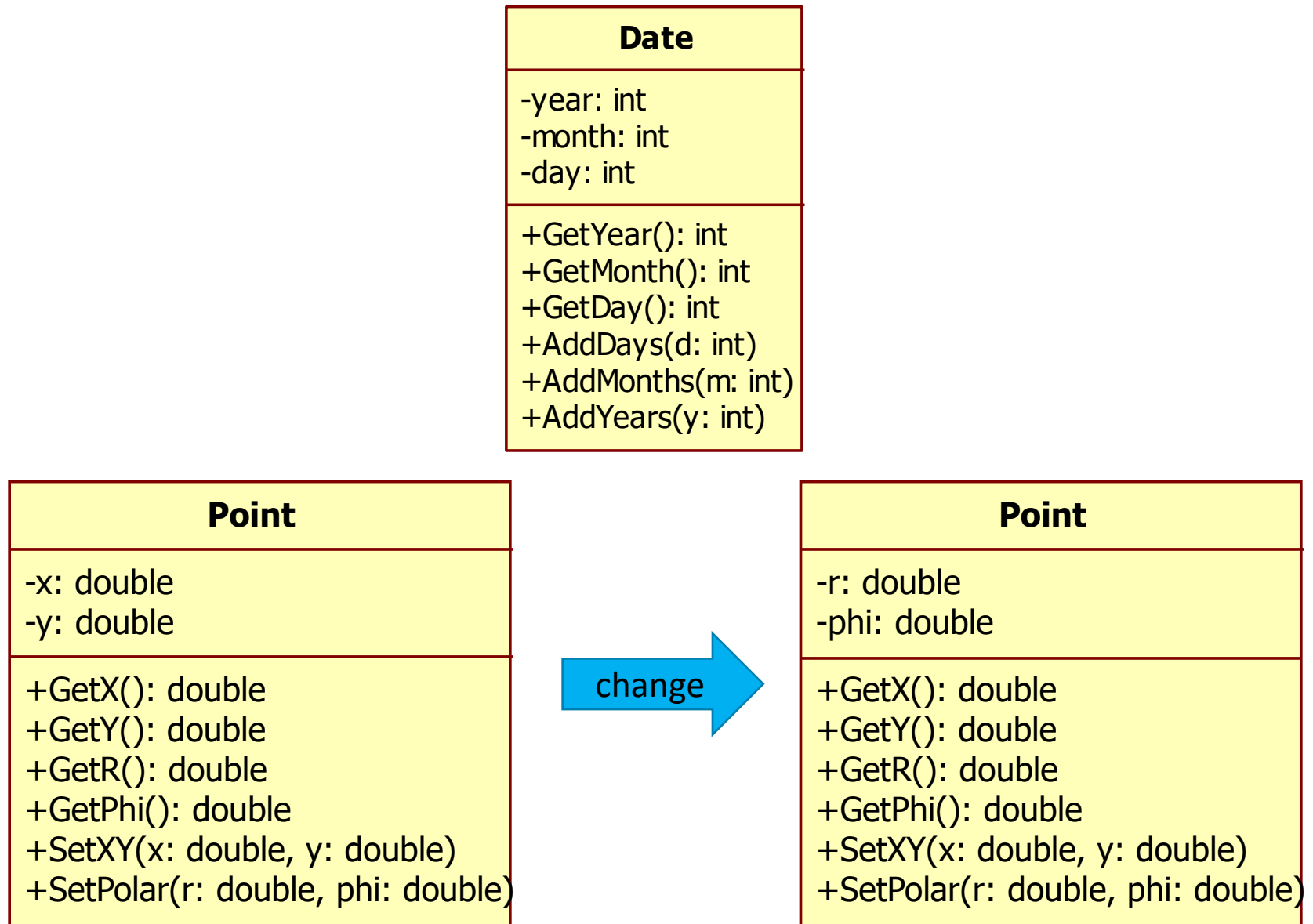
C1. Attributes should be private

- Problems if violated:
 - protected and public attributes violate information hiding
 - maintenance issues if data representation changes
 - constraints and invariants 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.Write(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.Write(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](#), [addHierarchyChangeListener](#), [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](#), [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](#), [list](#), [list](#), [locate](#), [location](#), [lostFocus](#), [minimumSize](#), [mouseDown](#), [mouseDrag](#), [mouseenter](#), [mouseleave](#), [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](#), [removeHierarchyChangeListener](#), [removeInputMethodListener](#), [removeKeyListener](#), [removeMouseListener](#), [removeMouseMotionListener](#), [removeMouseWheelListener](#), [removeNotify](#), [removePropertyChangeListener](#), [removePropertyChangeListener](#), [repaint](#), [repaint](#), [repaint](#), [repaint](#), [requestFocus](#), [requestFocus](#), [requestFocusInWindow](#), [requestFocusInWindow](#), [reshape](#), [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

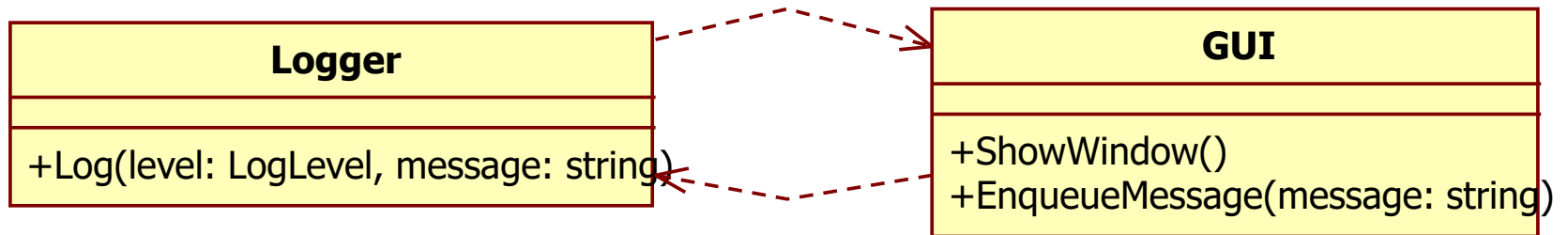
- Useful methods:
 - equality testing, printing, parsing, deep copy
 - C#: ToString(), Equals(), GetHashCode()
 - Java: toString(), equals(), hashCode()
 - C++: copy constructor, operator==, operator=, operator<< to an ostream
- These are useful since the developer can expect these to work
- They can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

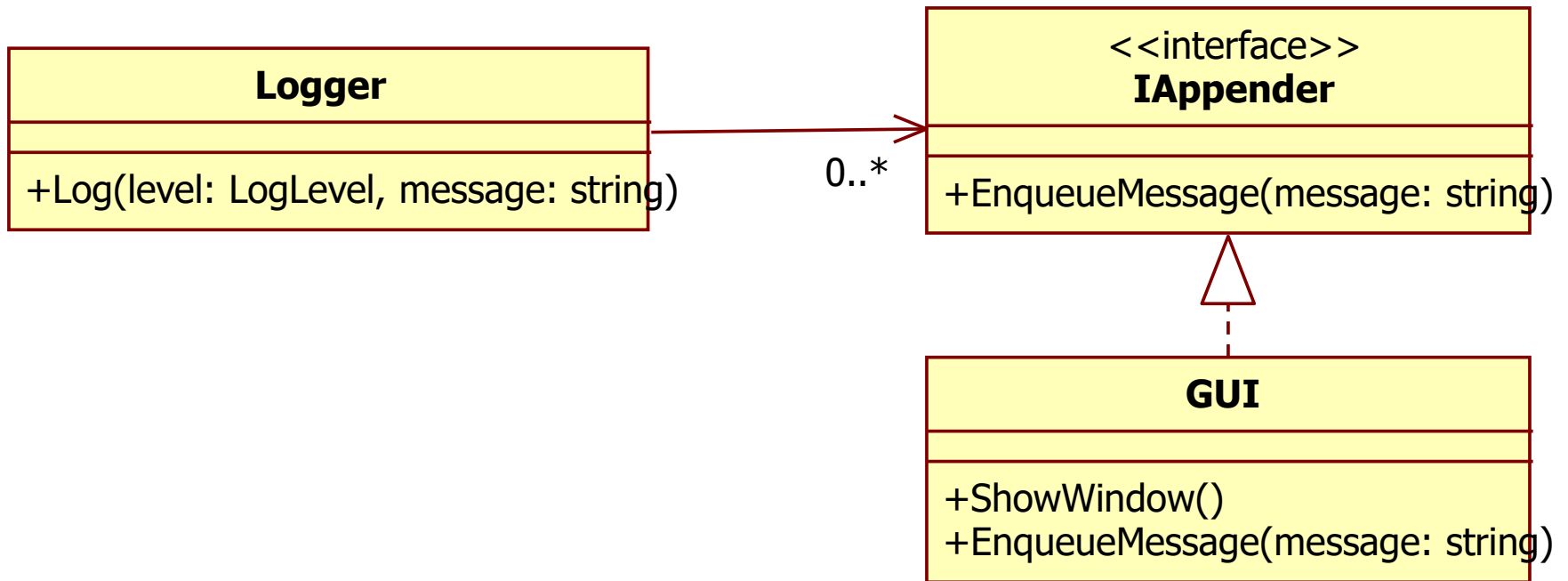
- Problems if violated:
 - if a class depends on its users it cannot be reused without them
 - descendants already depend on the base class
 - if a class depends on its descendants it has to depend on later added descendants, too
 - this violates OCP and LSP
 - also this creates circular dependencies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

C5. A class should not depend on its users

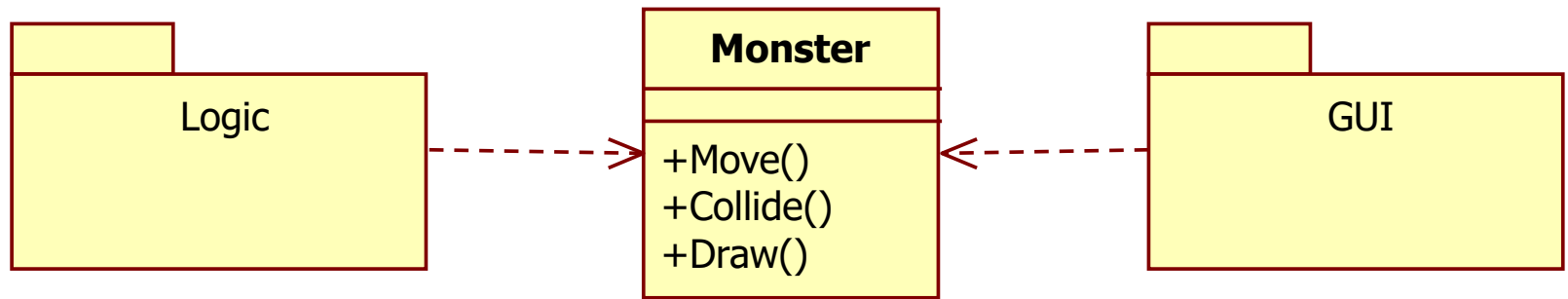
- Rule:

- 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

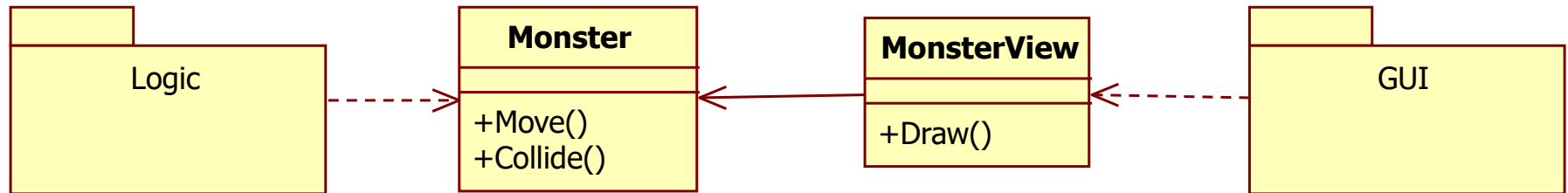
- Rule:

- 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

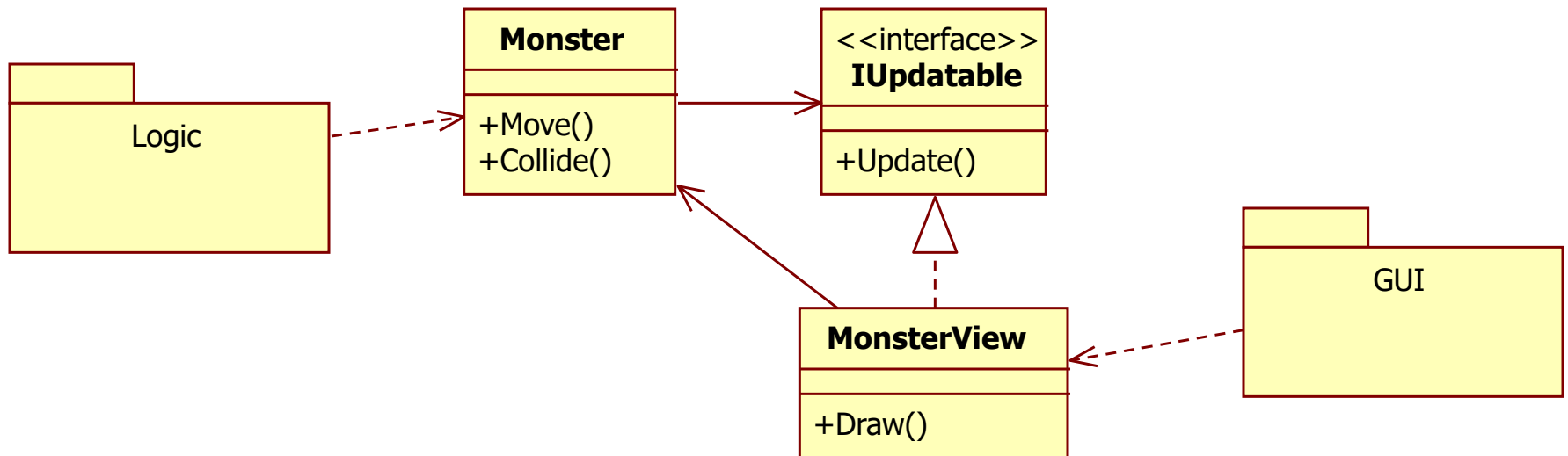
- Exception:

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

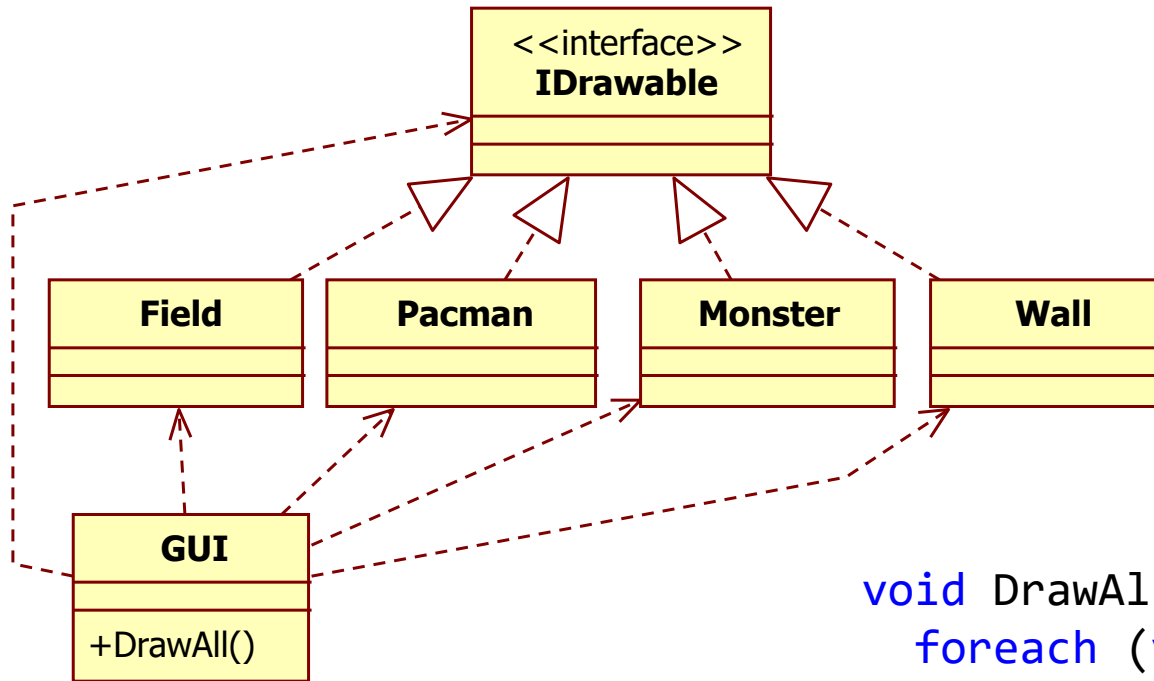
C6. Solution I.



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

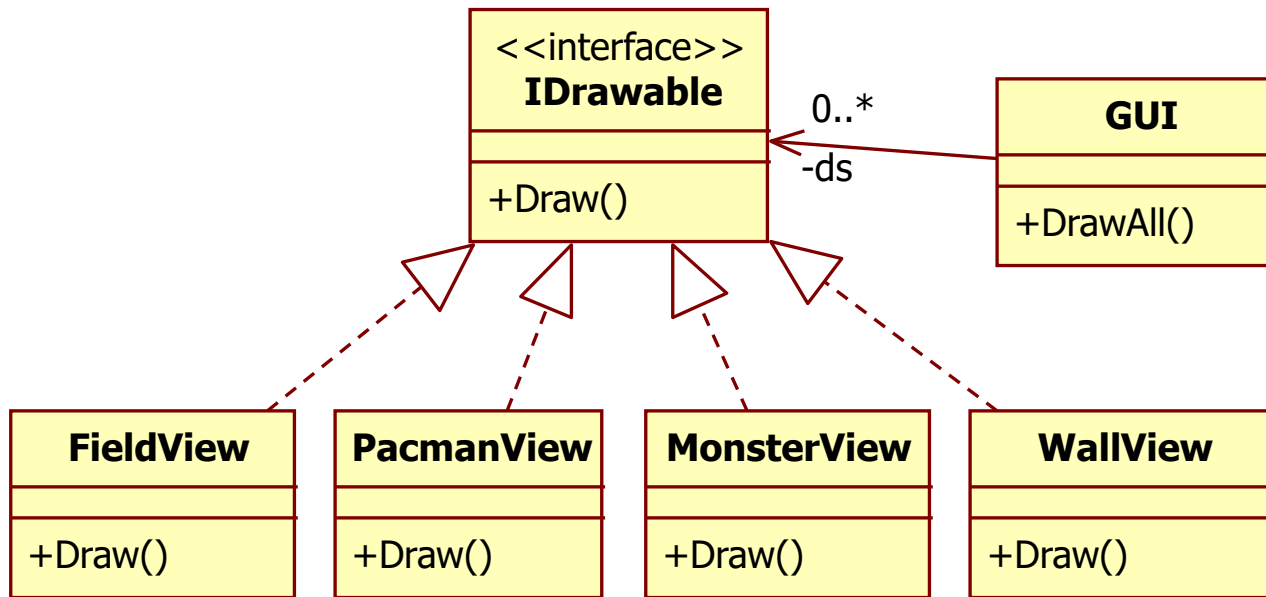
- Rule:

- 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

Game
<ul style="list-style-type: none">-logMessages: LogMessage[]-highScore: HighScoreEntry[]-highScoreWindow: Window-timer: Timer-gameWindow: Window
<ul style="list-style-type: none">+Log(level: LogLevel, message: string)+AddToHighScore(score: int, name: string)+DisplayHighScore()+Step()+DisplayGameWindow()+Draw()

Problem: too many and unrelated responsibilities

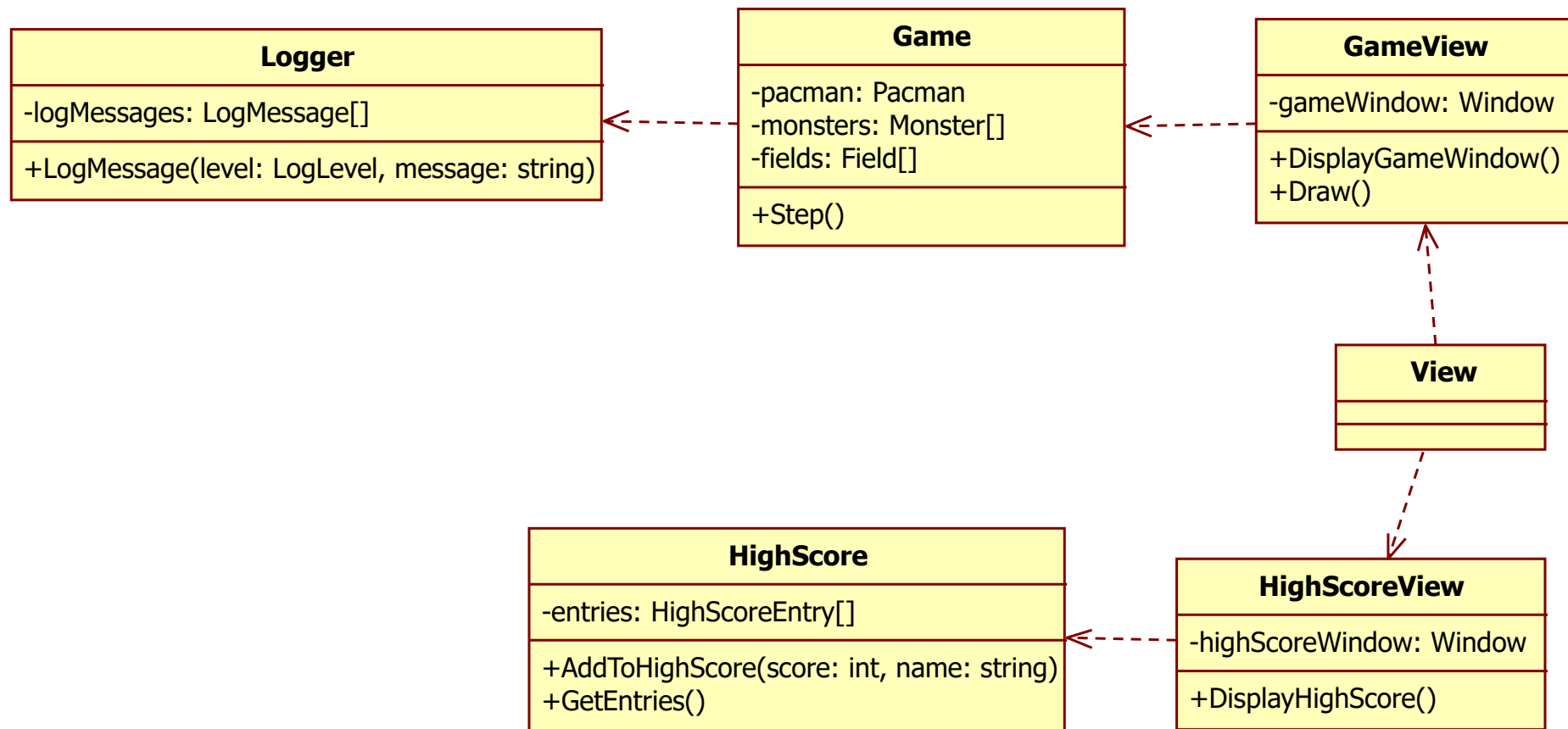
C8. Methods should use most of the members in its class

- 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

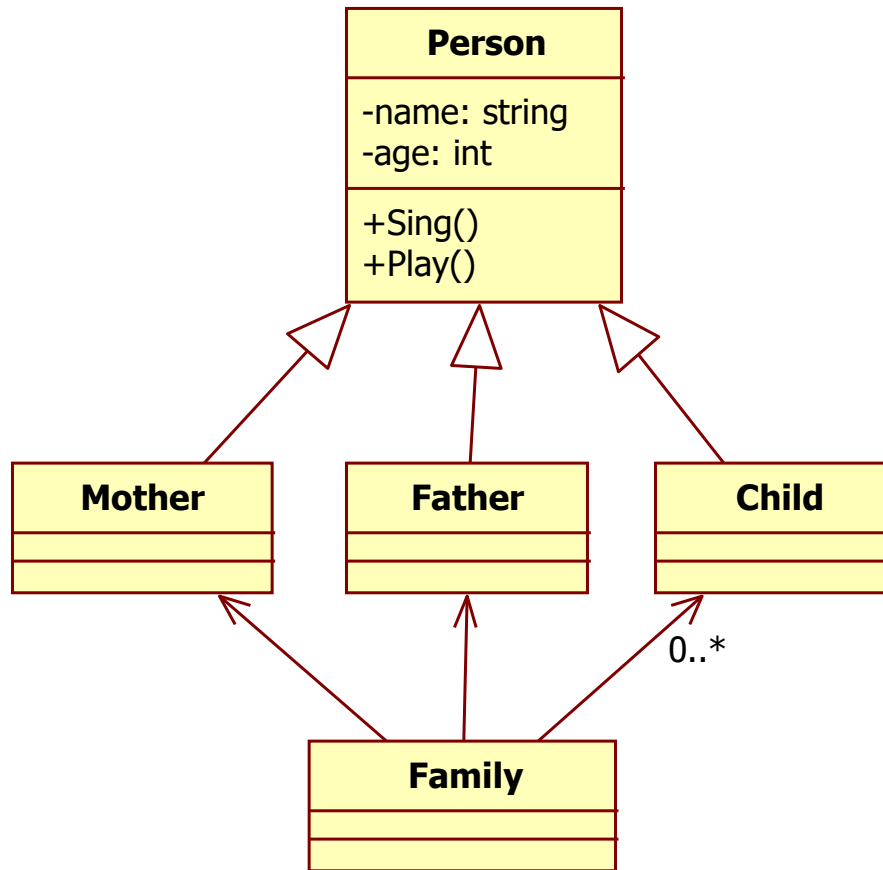
C8. Methods should use most of the members in its class

- 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

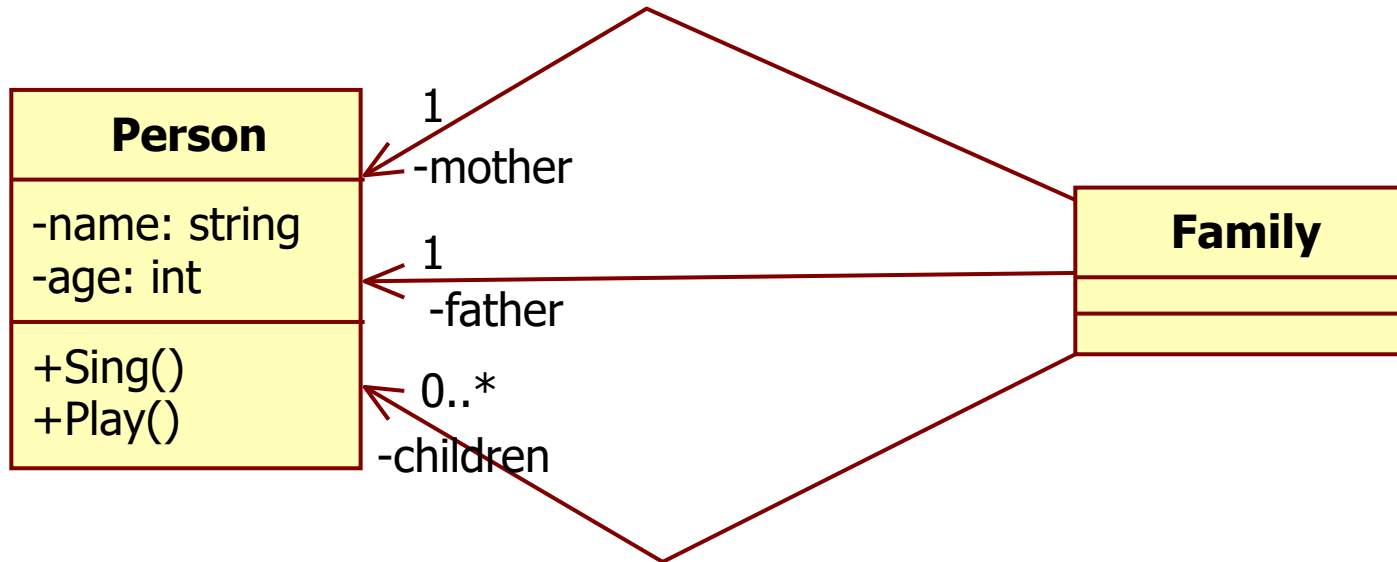
- Rules:

- 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

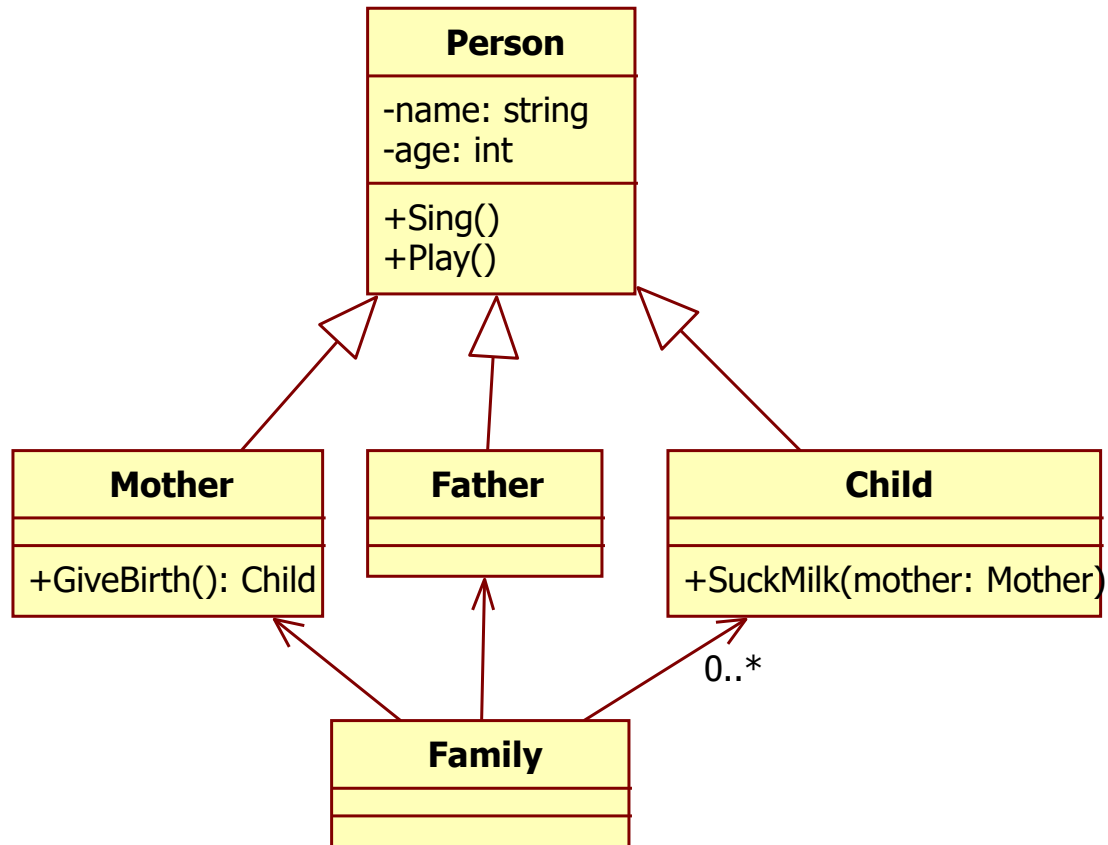
- Exceptions:

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

C9. Solution I.



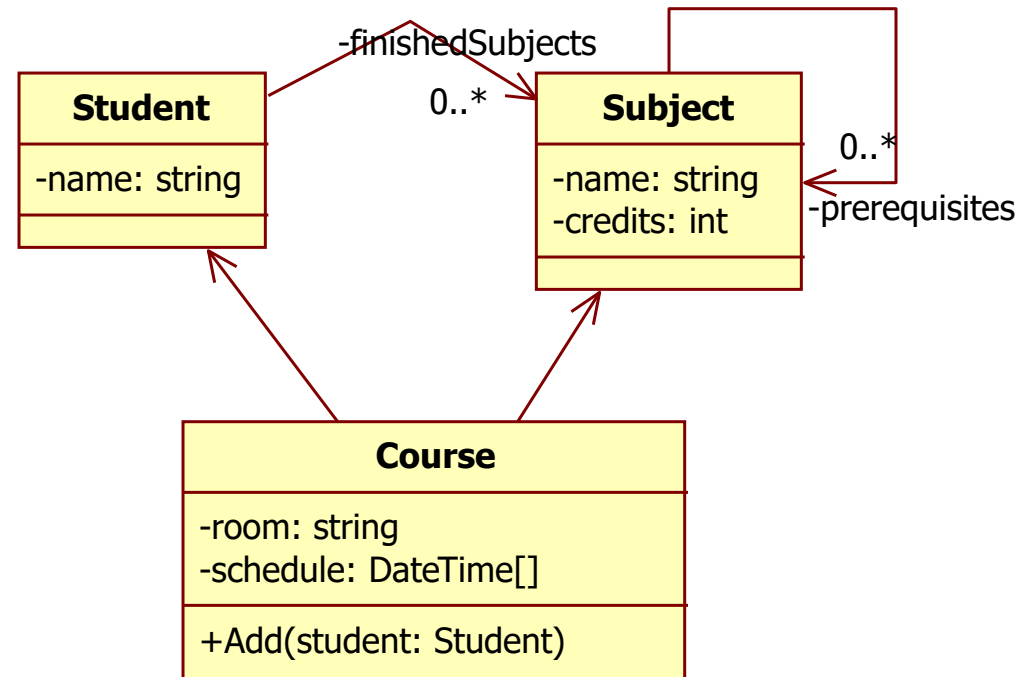
C9. Solution II.



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

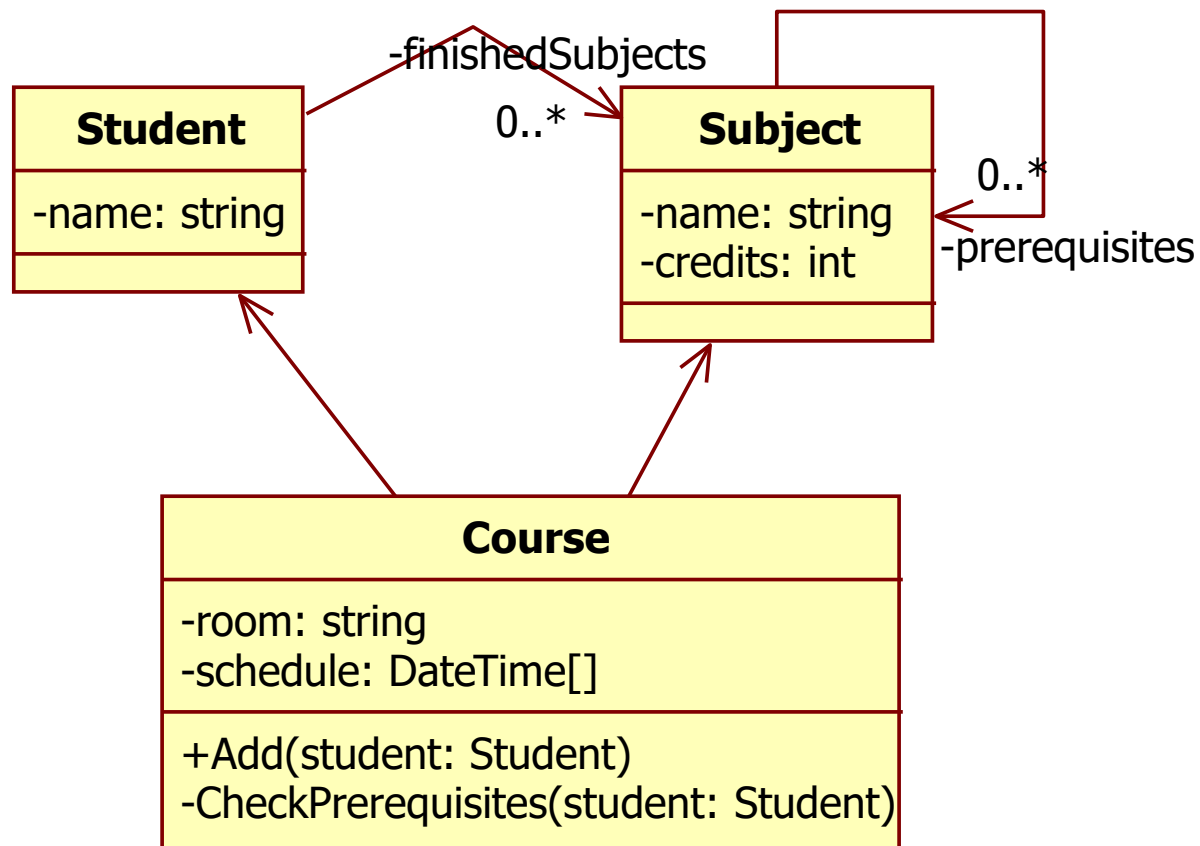
- Problems if violated:
 - some classes have too much responsibility, some have too little or none
 - if responsibility is not divided evenly, it usually shows as behavior god classes and classes with only accessor methods
 - usually unevenly distributed responsibility results from the wrong design process: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

R1. Distribute responsibility horizontally evenly

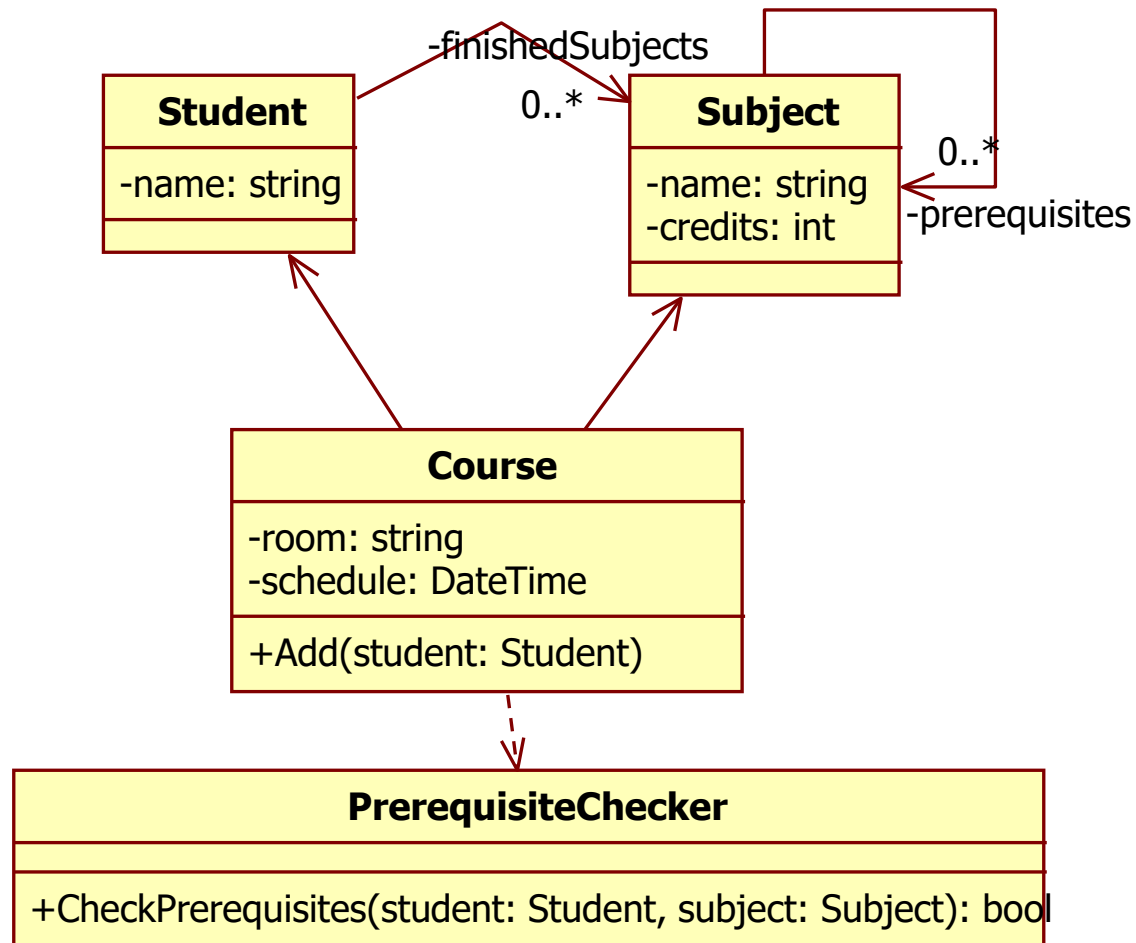
- Rules:

- 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

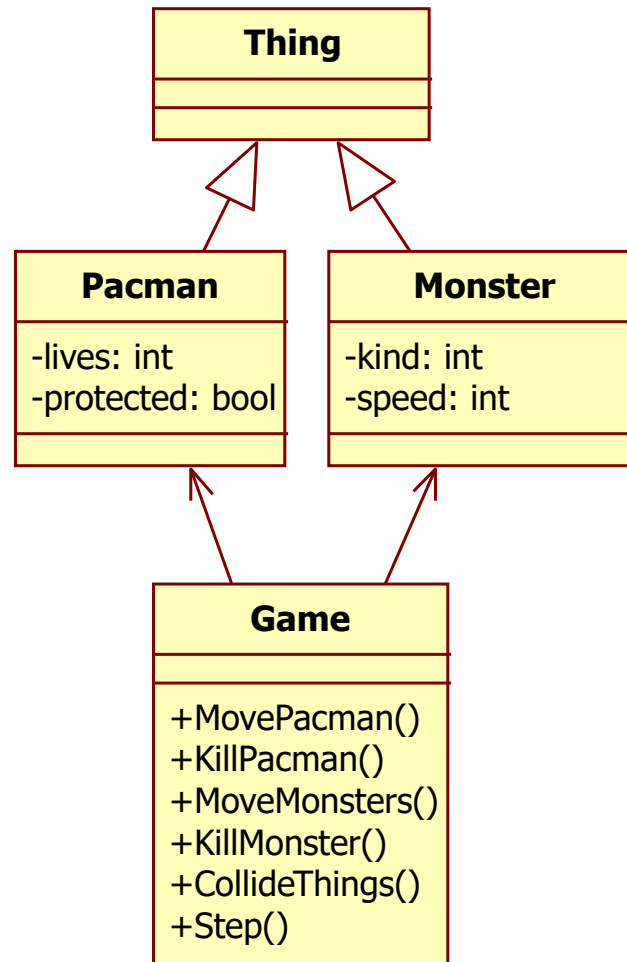
R1. Solution I.



R1. Solution II.



R2-R3. Problem



Problem: a god class + a lot of data classes

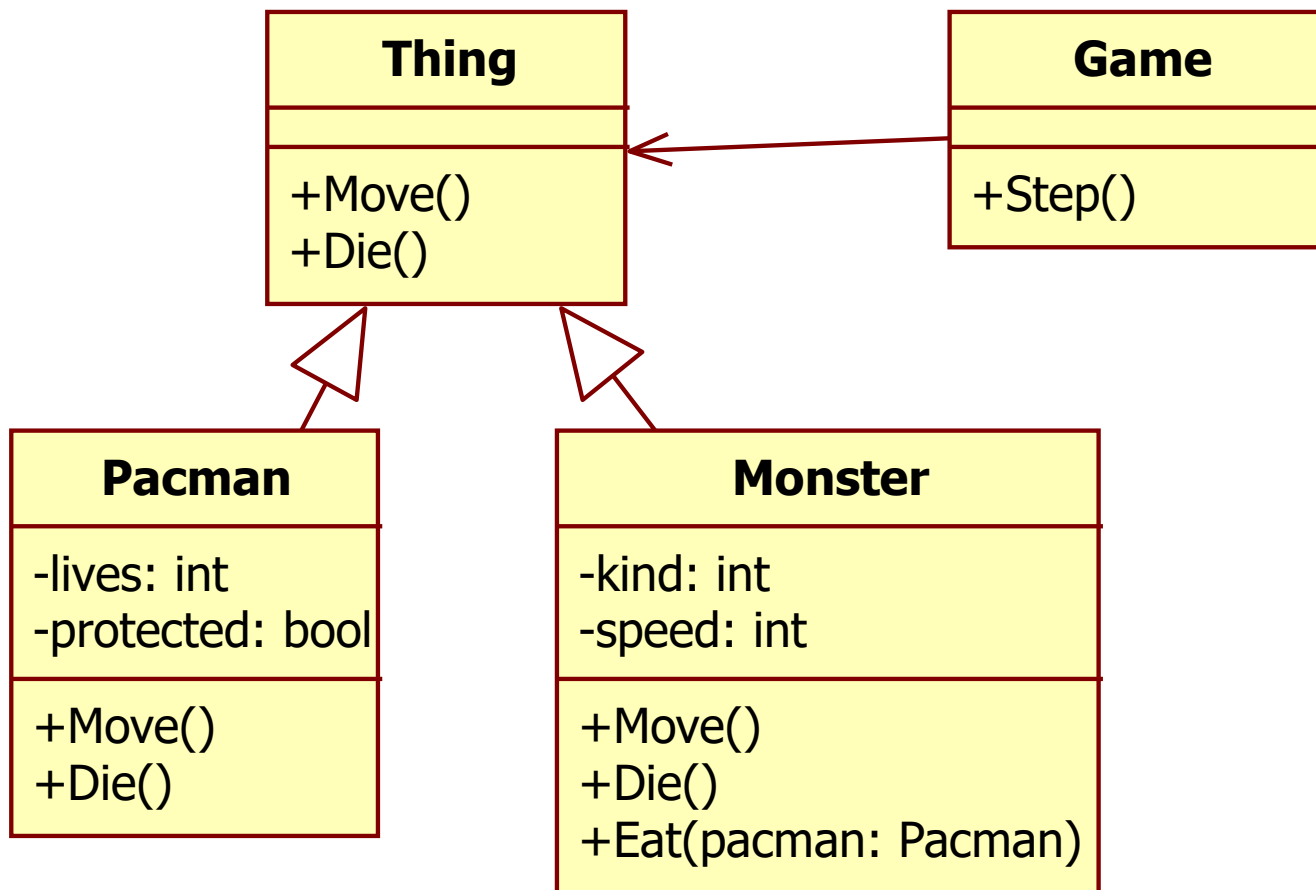
R2. Avoid god classes

- 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.

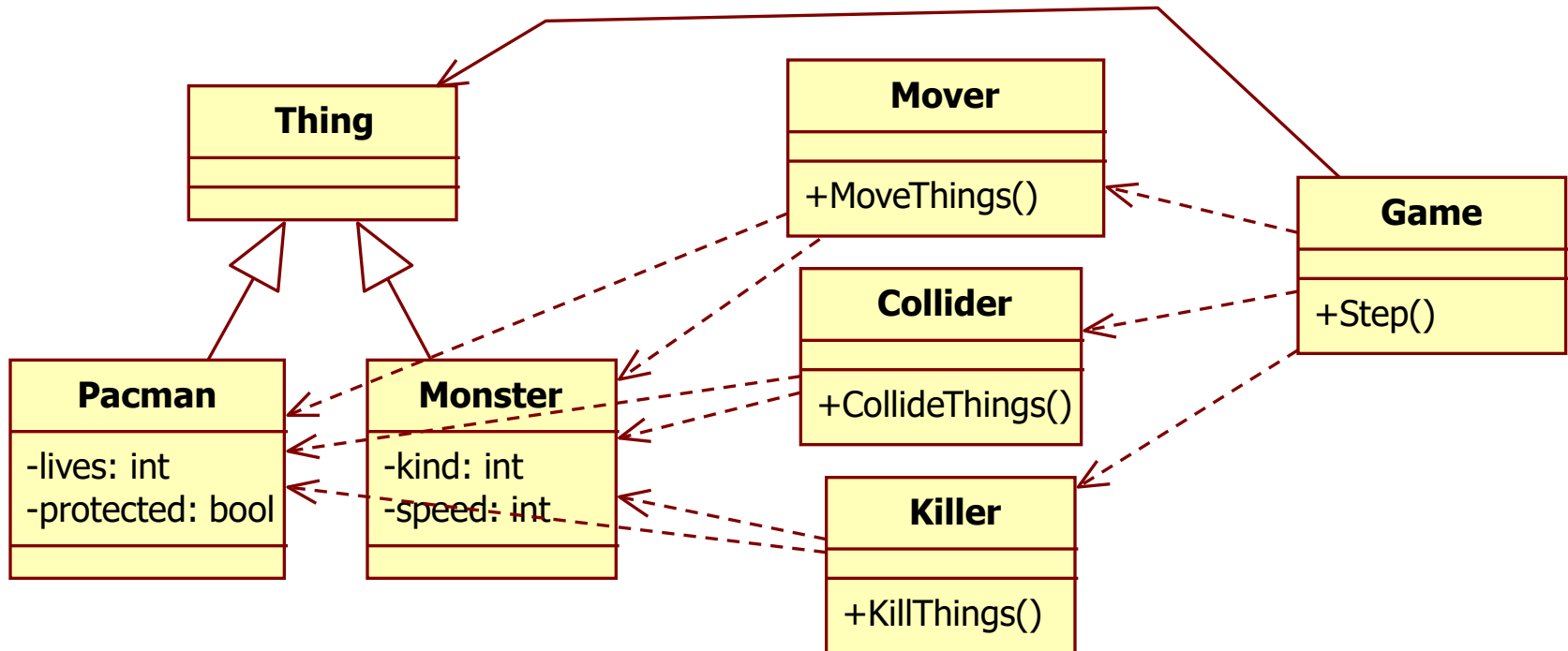
R3. Avoid classes with only accessor methods

- 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

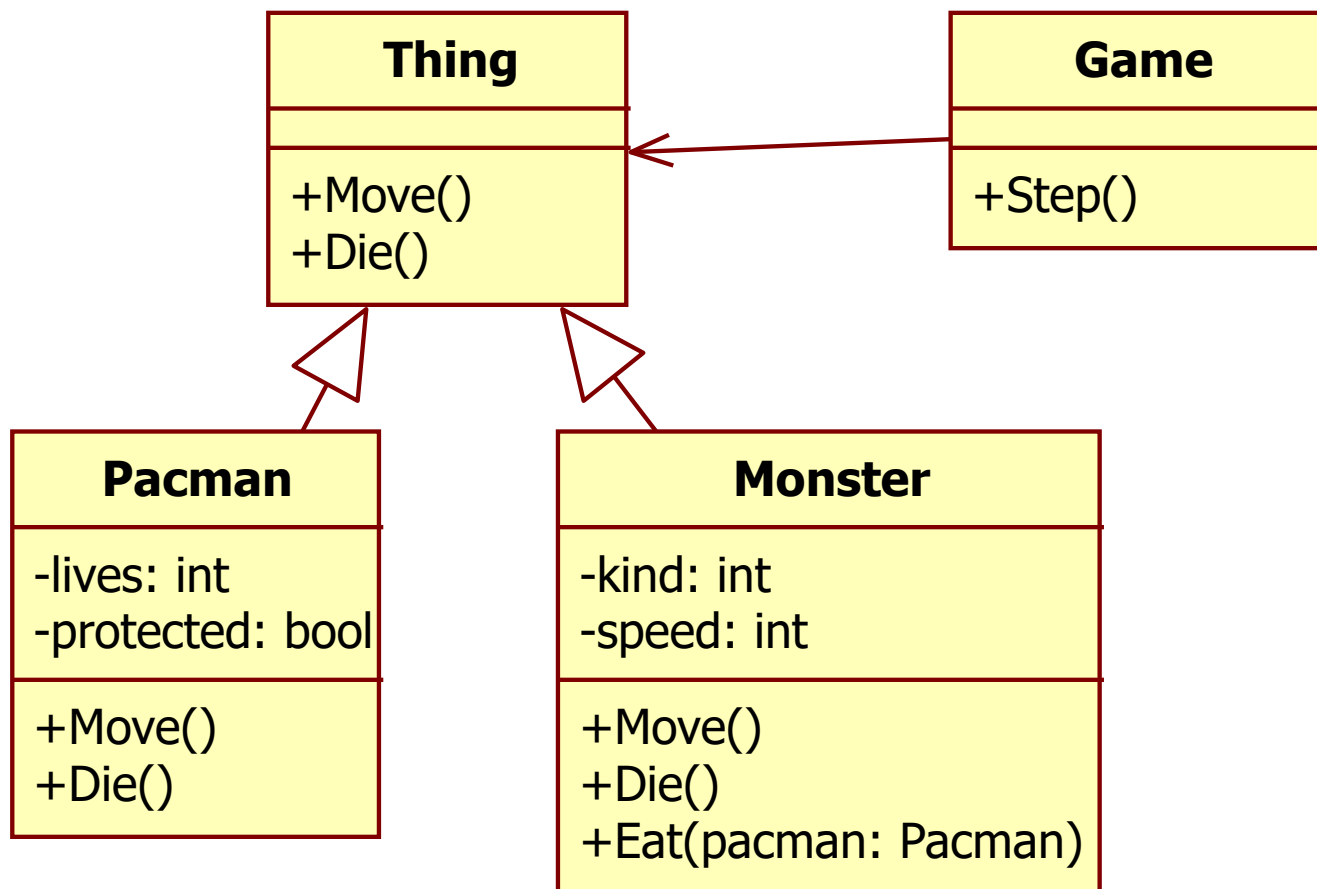


Problem: a lot of classes with a single method

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

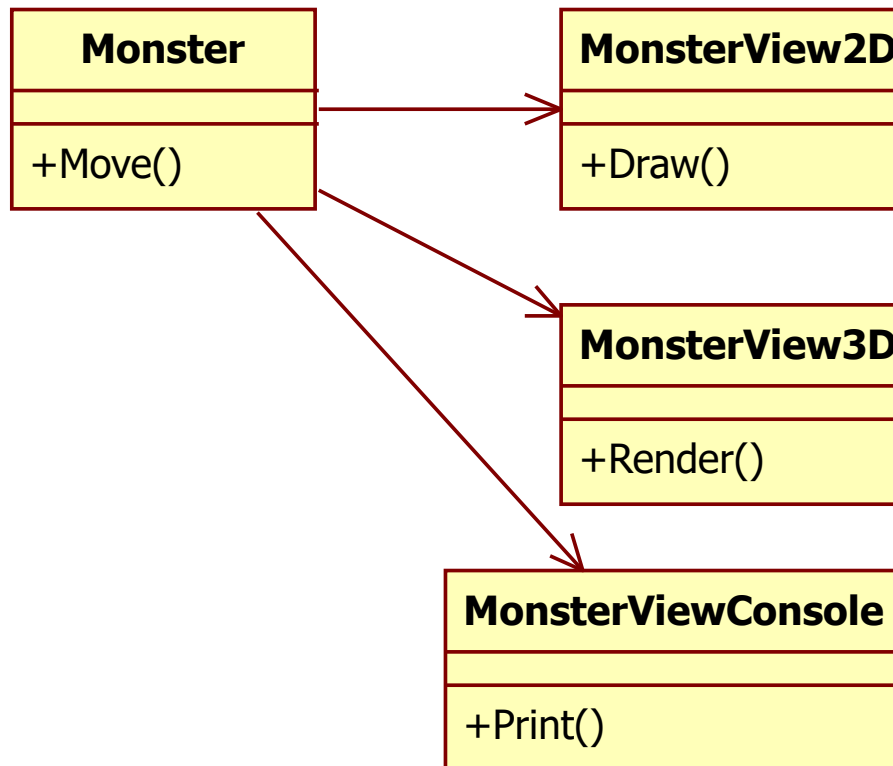
R6. Model at the appropriate abstraction level

- 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

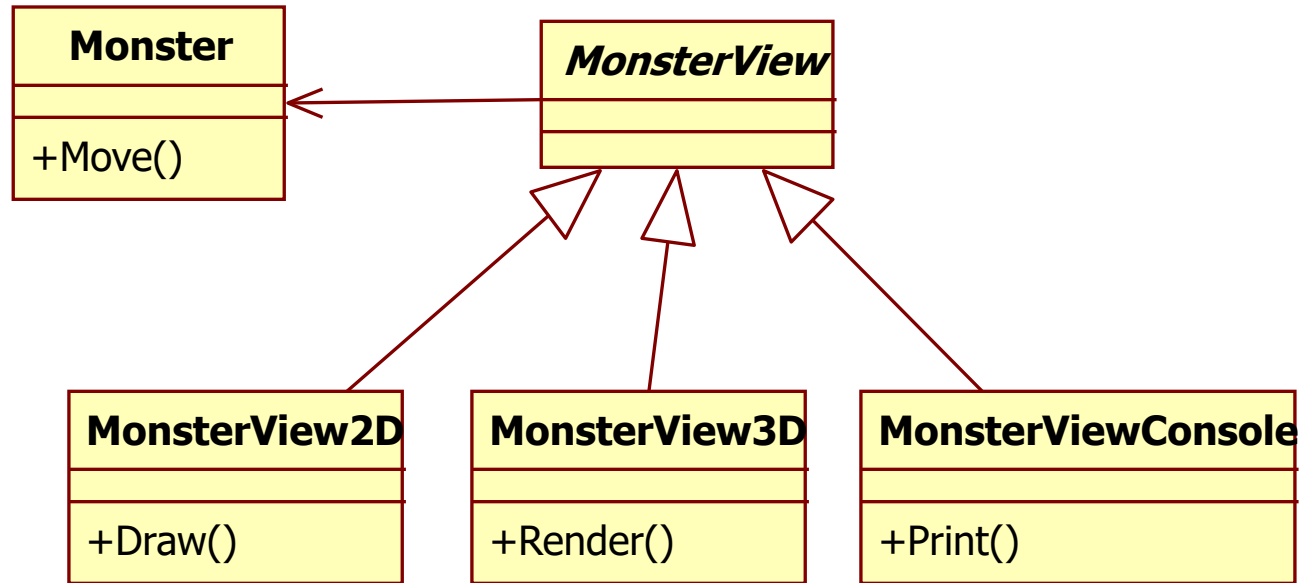
■ Problems if violated:

- 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

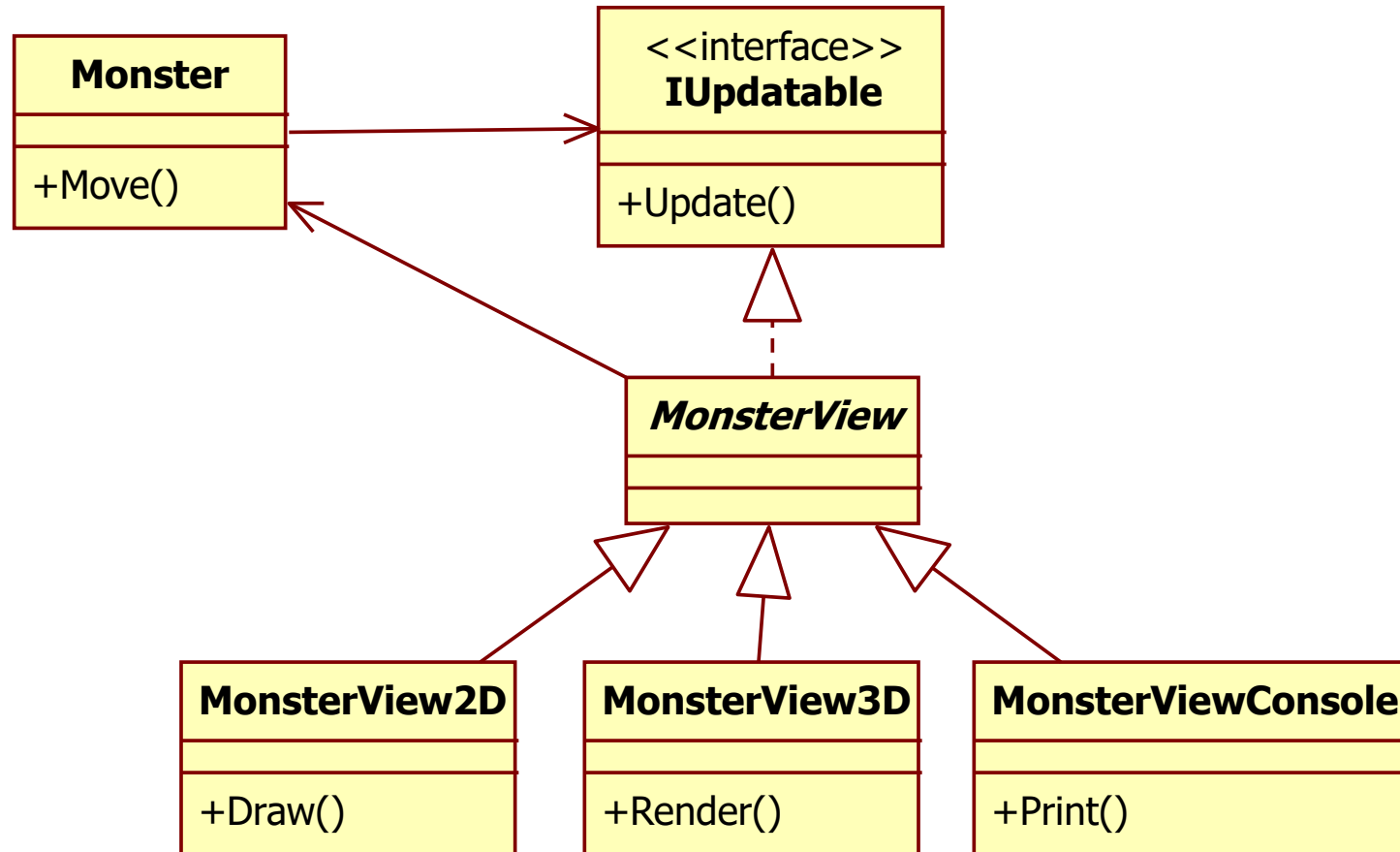
■ Rules:

- 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

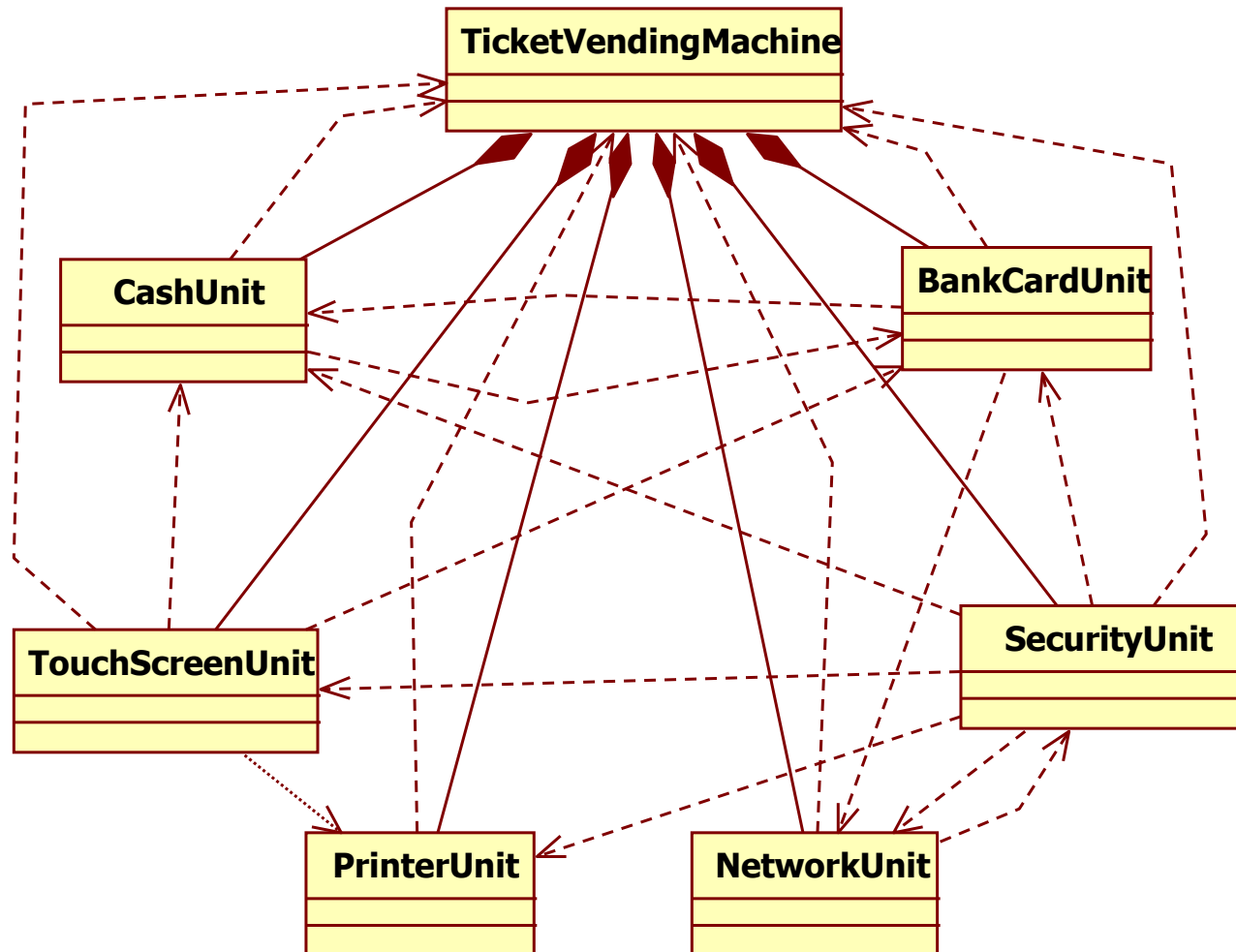


R8. Solution II. – push



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

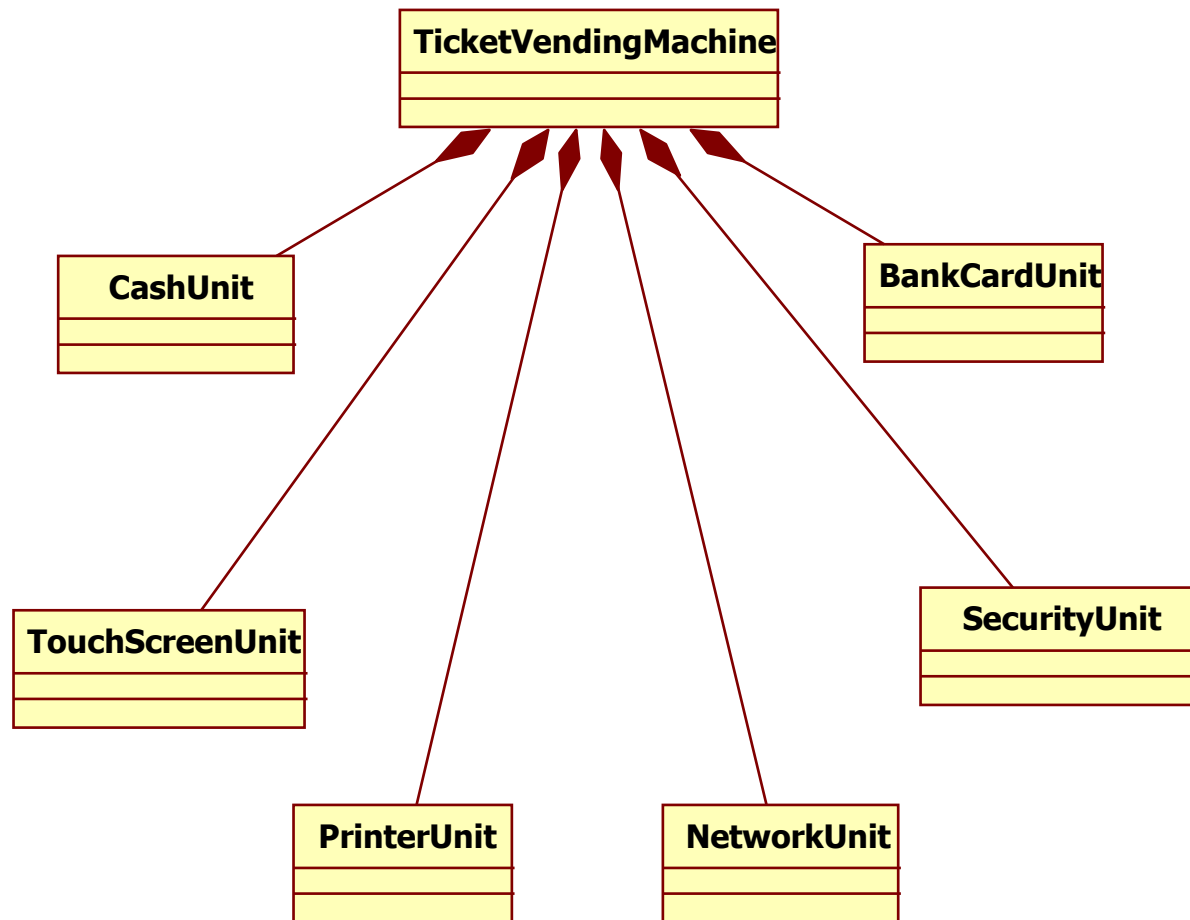
- Problems if violated:

- 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

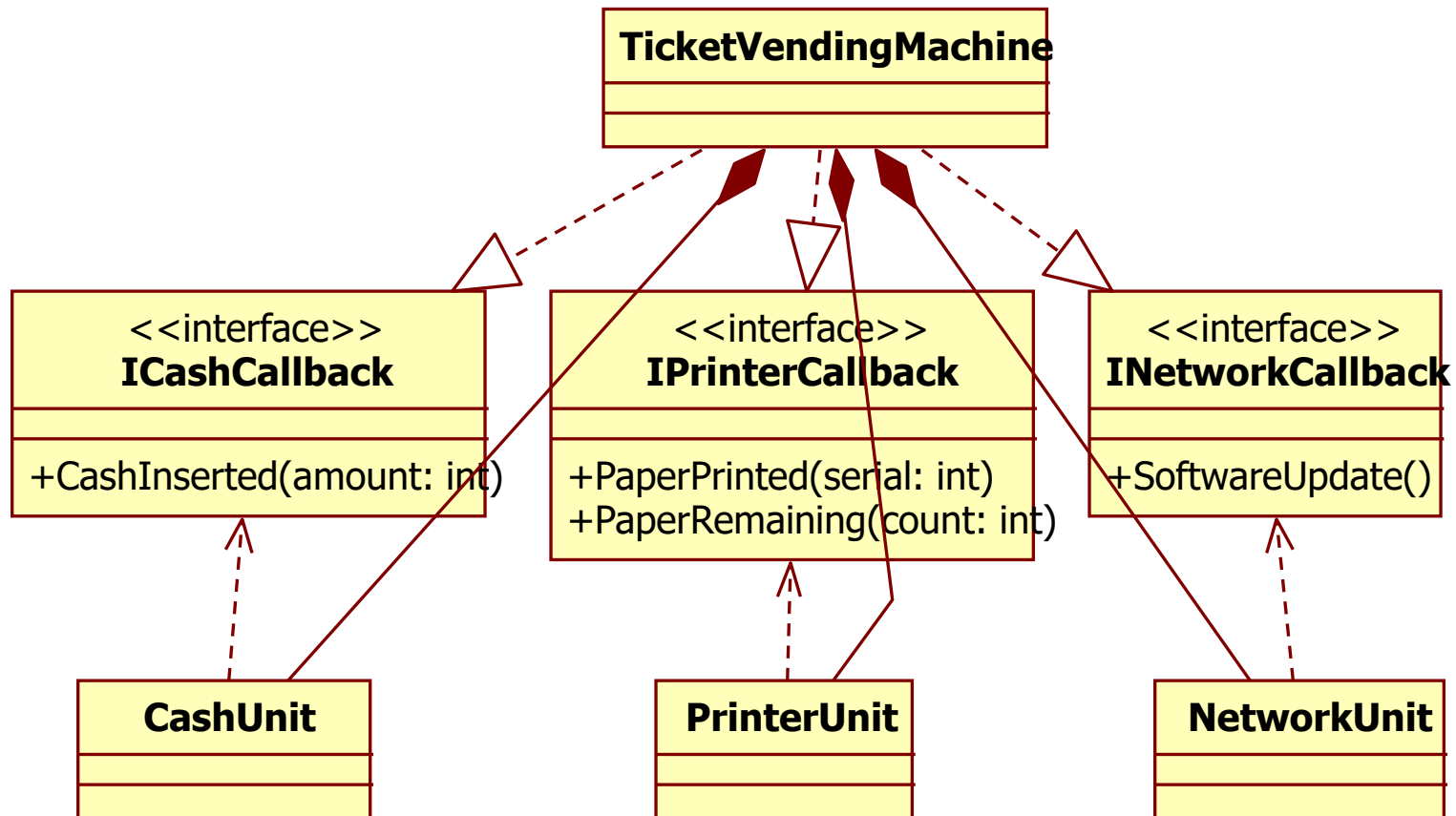
- Rules:

- 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.



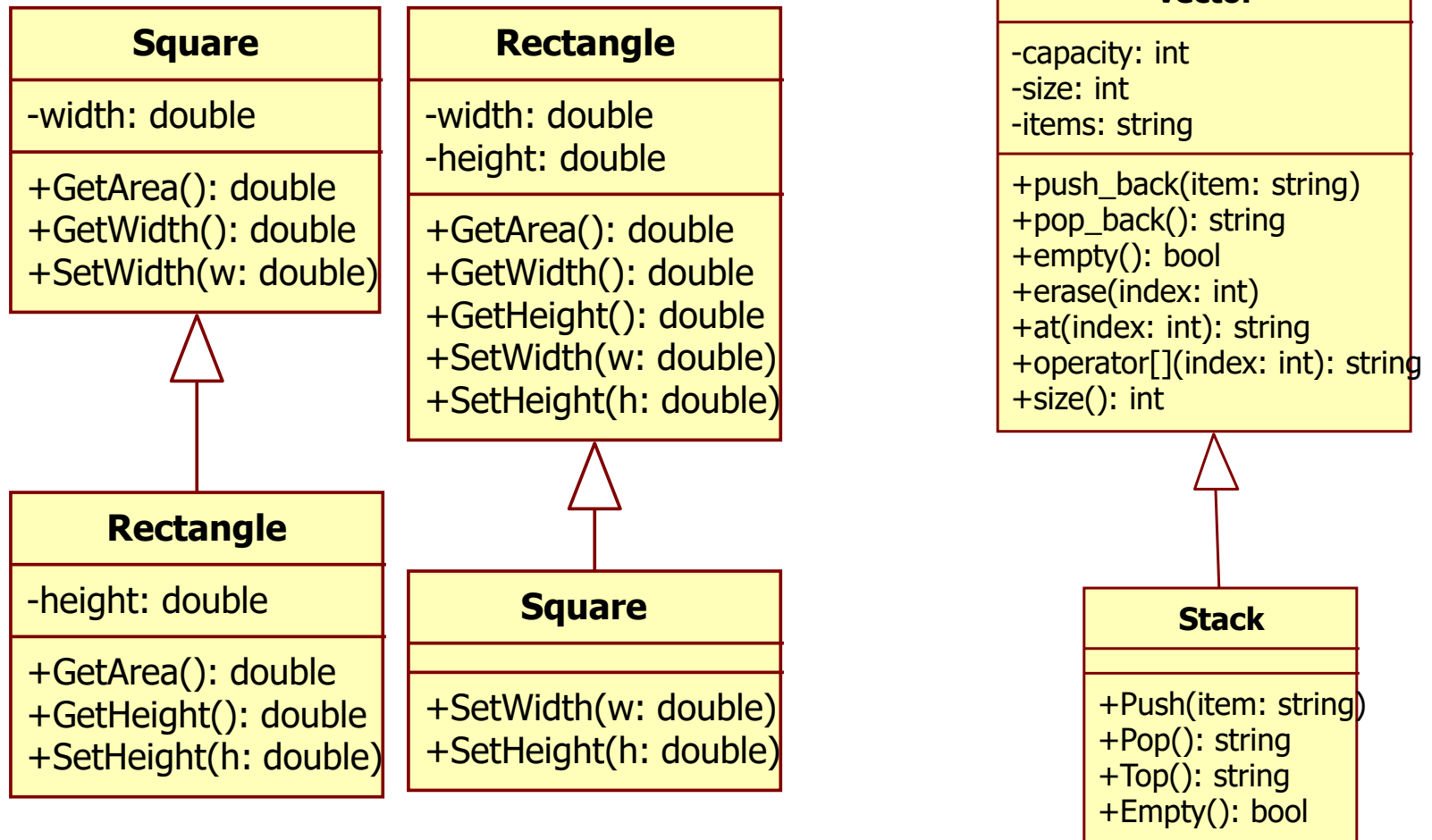
A1-7. Solution II.



...

Inheritance

11-2. Problem

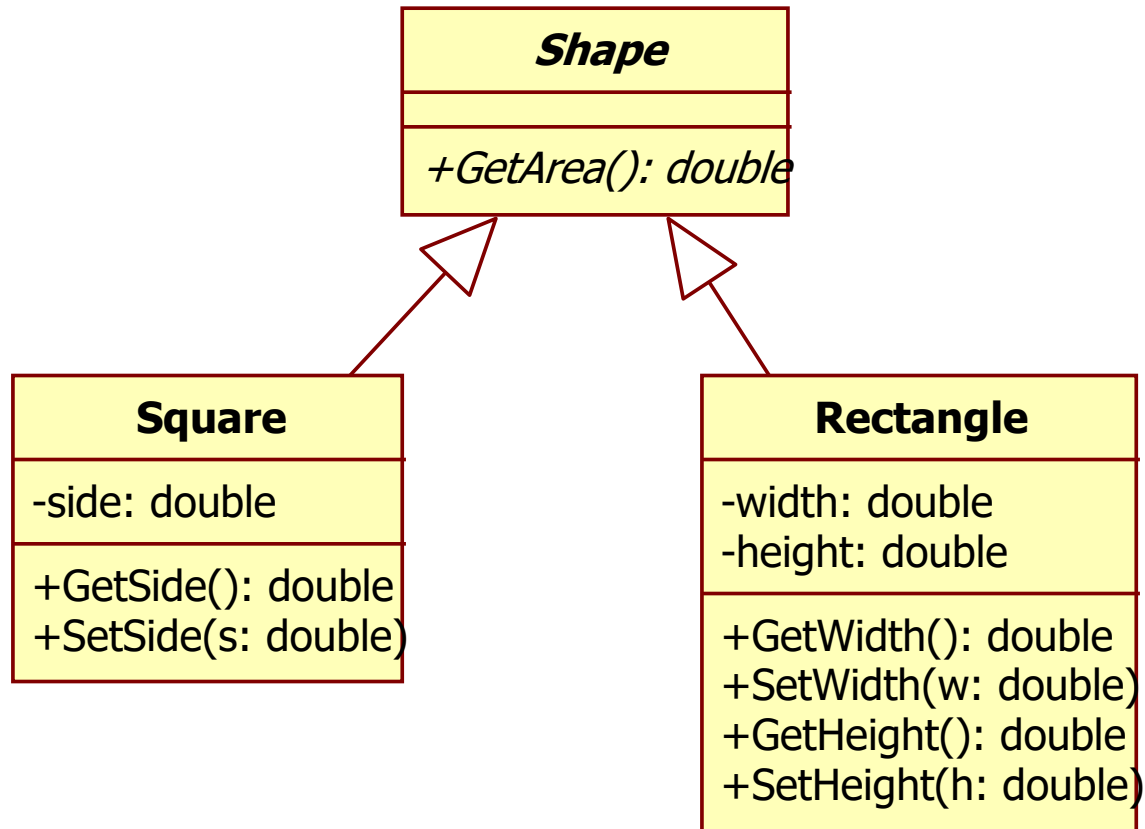


Problem: inheritance for data reuse

11. 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

1. Solution



12. Prefer containment over inheritance

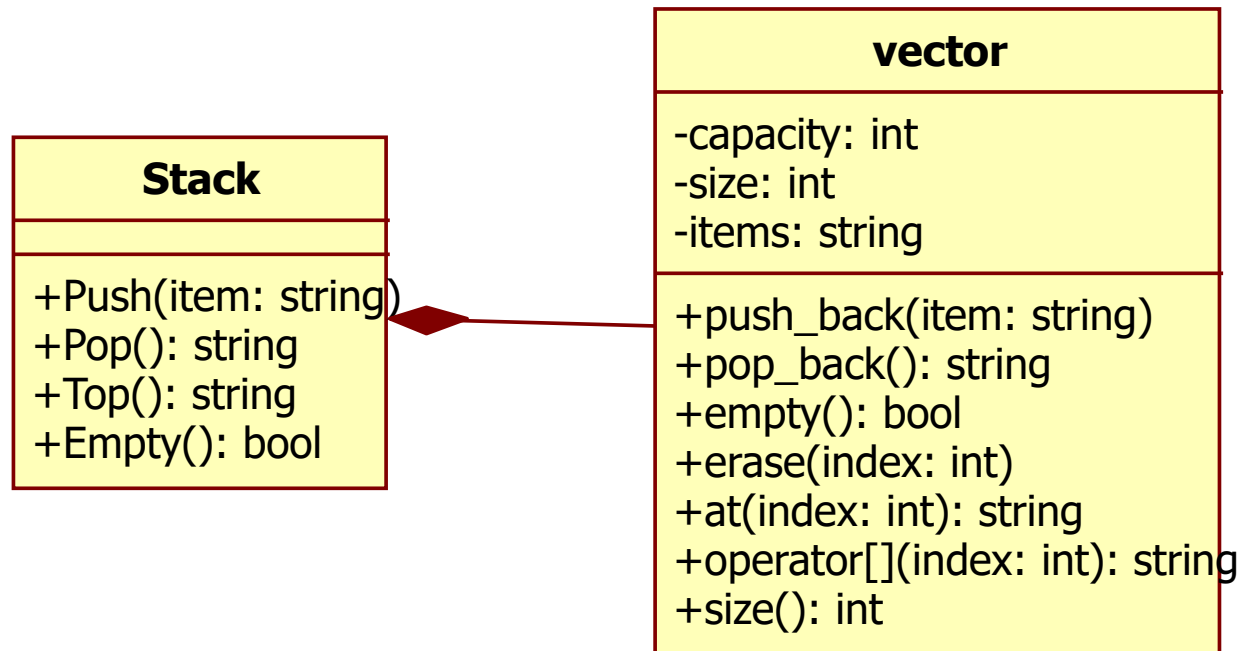
- Problems if violated:
 - inheritance is used for data reuse
 - exposing too much implementation details
 - public methods of the base class are also published
 - violation of LSP
 - inheritance is not black-box reuse
 - hard to change implementation later

12. Prefer containment over inheritance

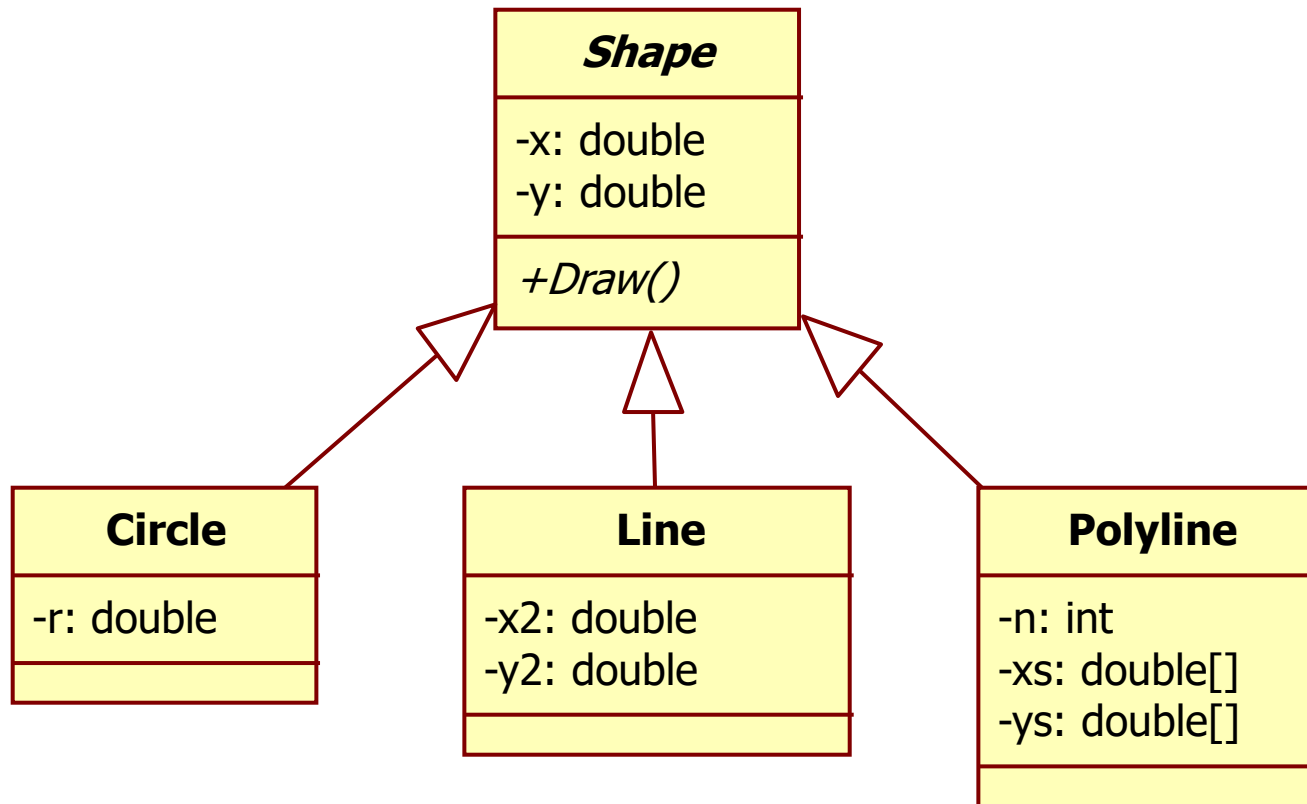
■ Rules:

- 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.

12. Solution



13. Problem

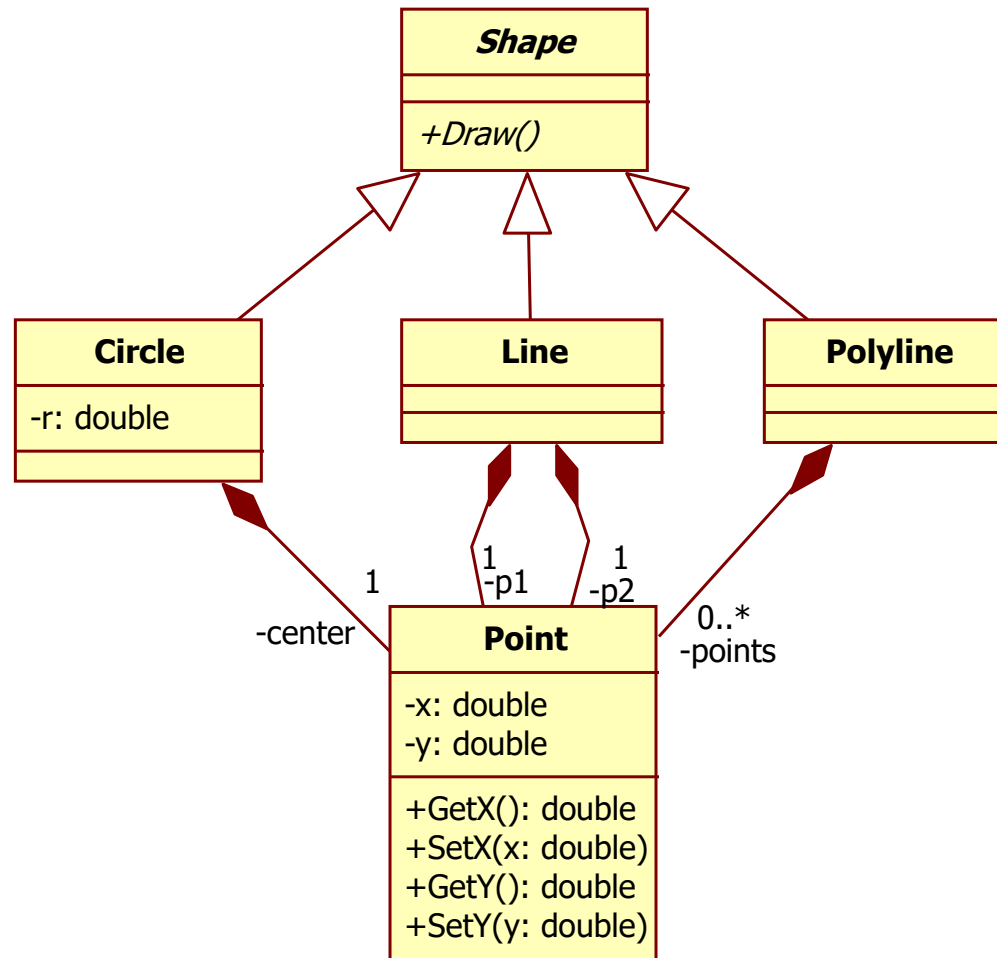


Problems: shared data without shared behavior, parallel arrays, special handling of data

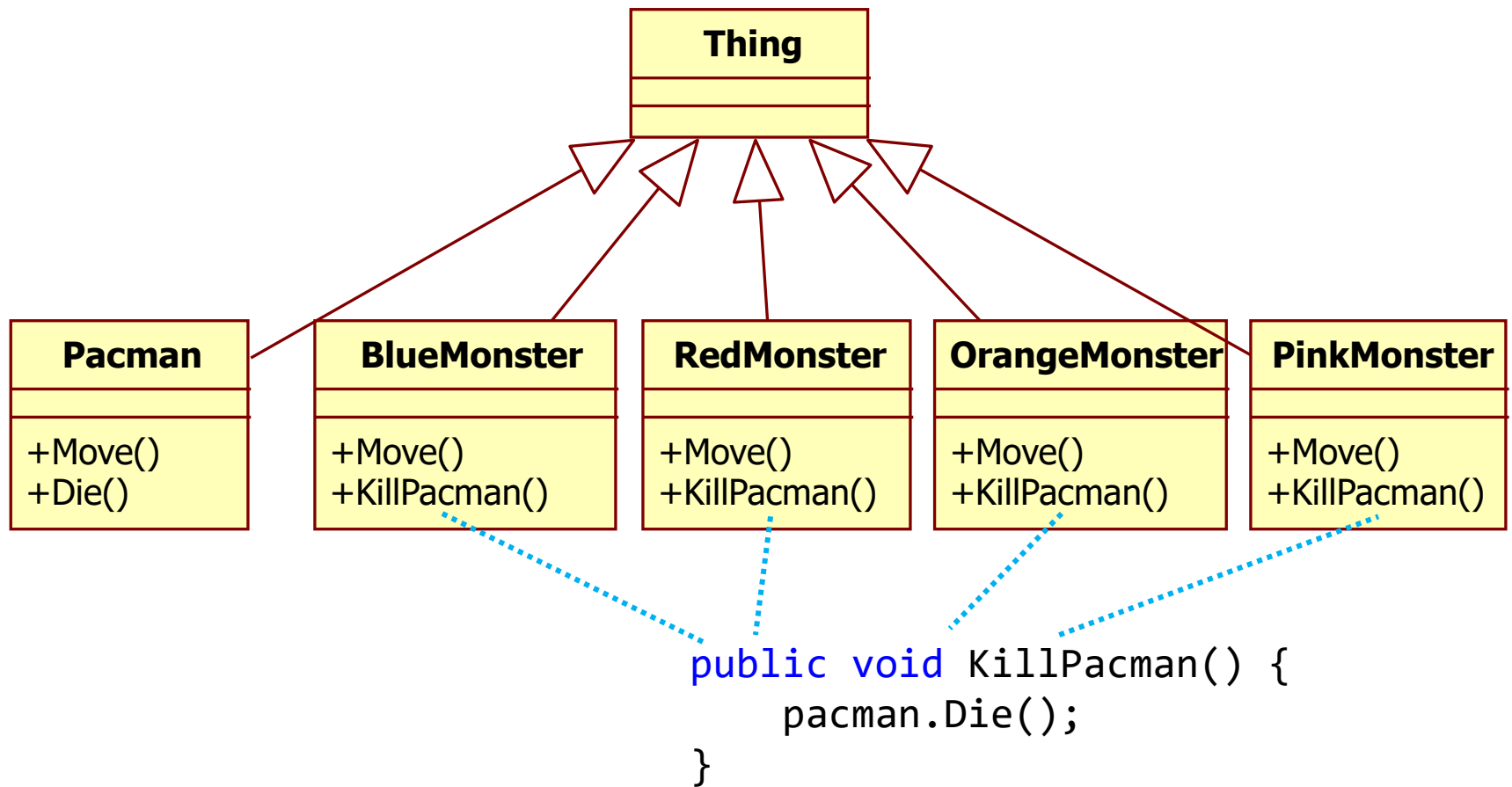
13. Shared data without shared behavior should be in a containment relationship

- 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

13. Solution



14-5. Problem



Problem: shared behavior repeated

14. 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

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

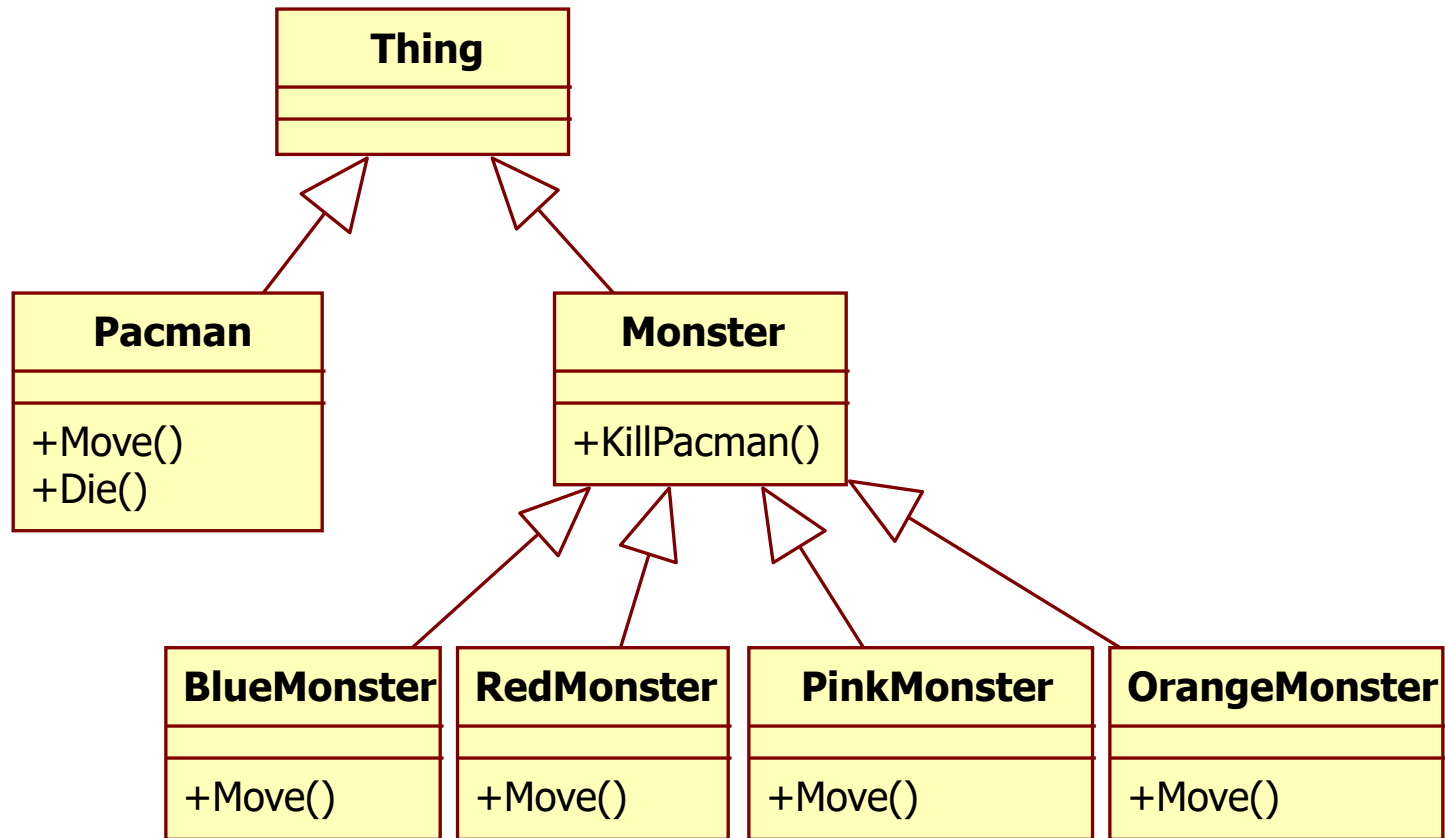
- Problems if violated:

- the common data and behavior may appear in multiple places violating DRY

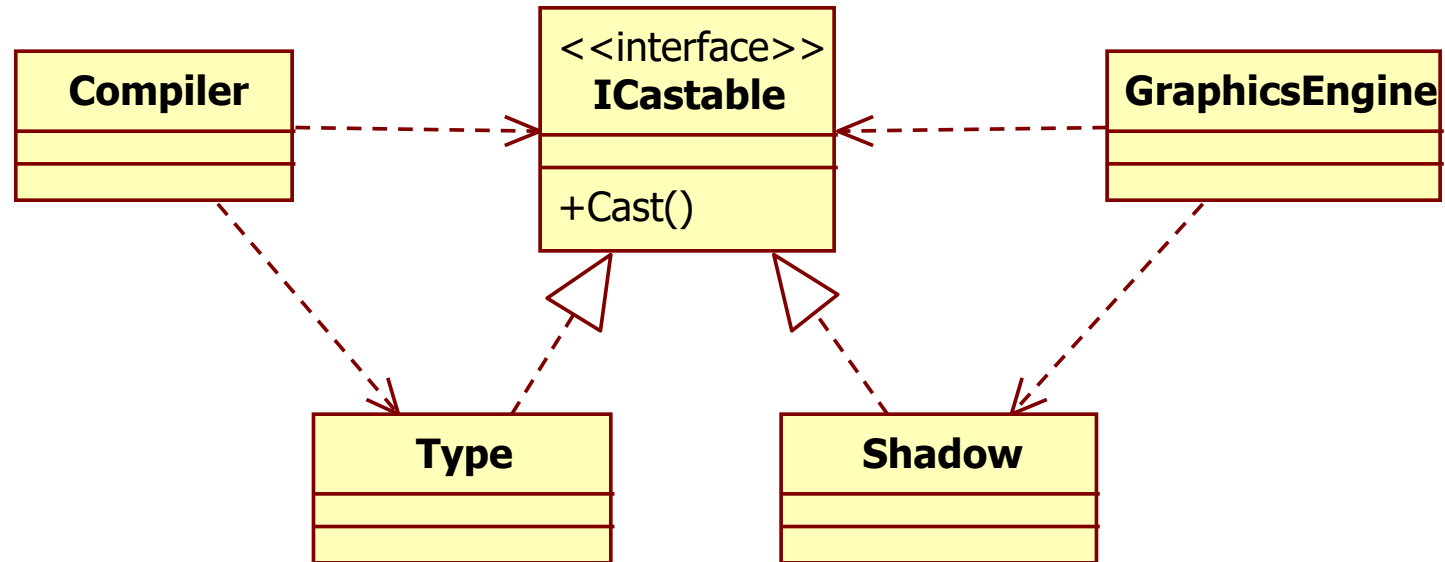
- Rules:

- 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

14-5. Solution



16. Problem



Problem: shared interface without similar behavior

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

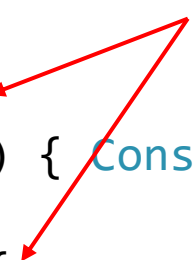
- Problems if violated:
 - violation of LSP: using an interface for an unintended purpose
- Rules:
 - a common interface should only be inherited if the descendants are used polymorphically, and they have similar behavior

Duck typing in C#

- Duck typing:


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

```
public class Duck {  
    public void Talk() { Console.WriteLine("Quack!"); }  
}  
public class Goose {  
    public void Talk() { Console.WriteLine("Honk!"); }  
}
```



No common ancestor or interface!

```
static void Speak(dynamic d) {  
    d.Talk();  
}
```



Still handled the same way

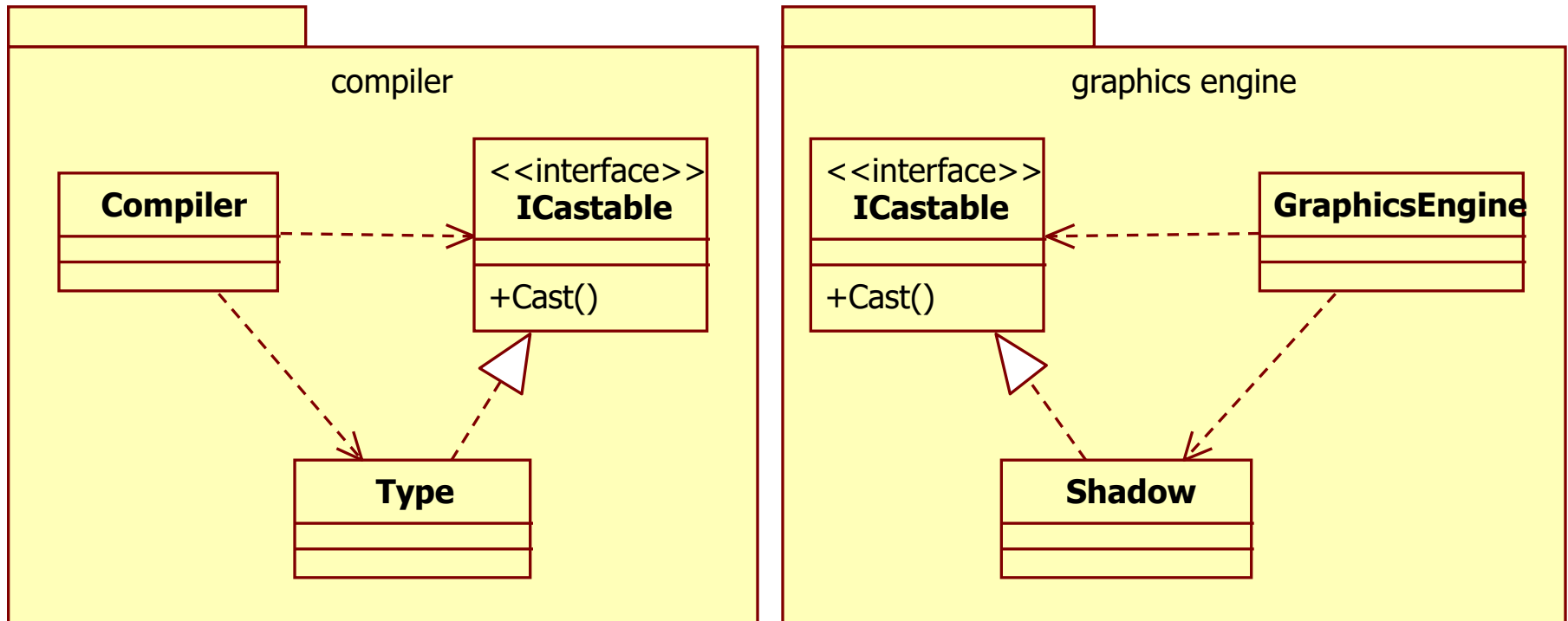
```
dynamic bird1 = new Duck();  
dynamic bird2 = new Goose();  
Speak(bird1);  
Speak(bird2);
```


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

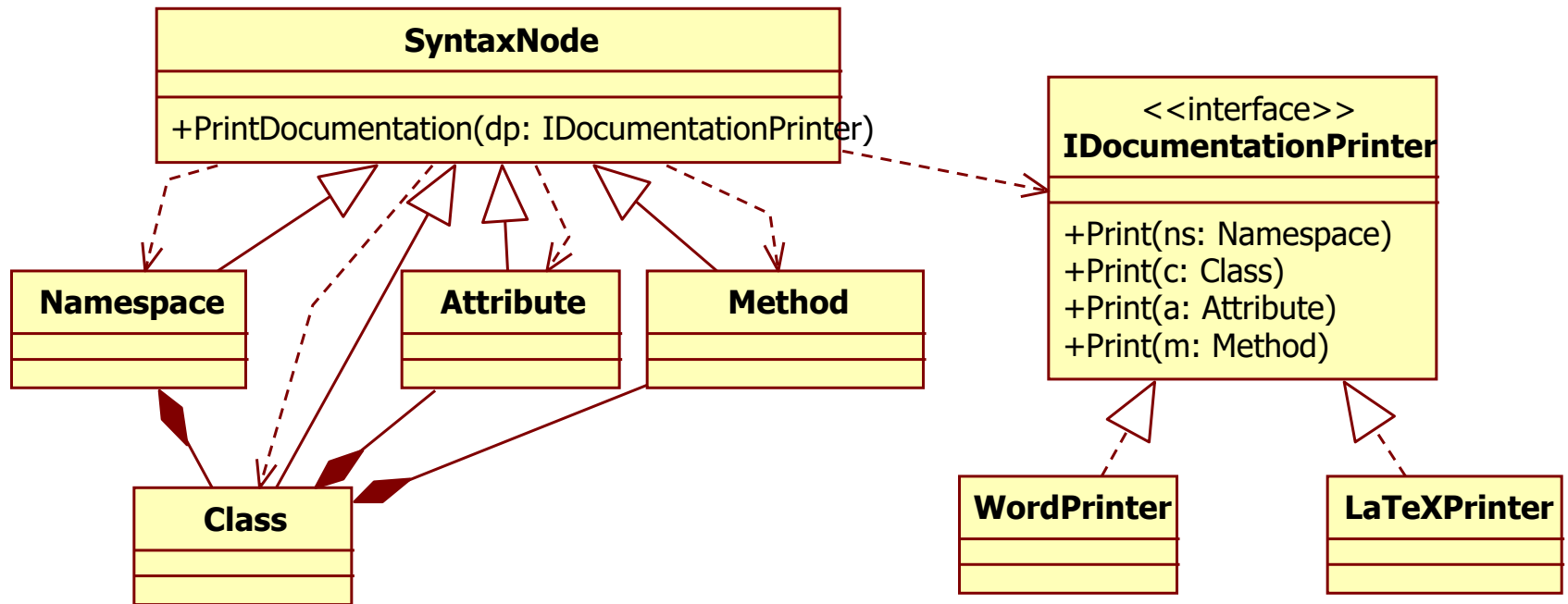
■ Rules:

- 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

16. Solution



17. Problem



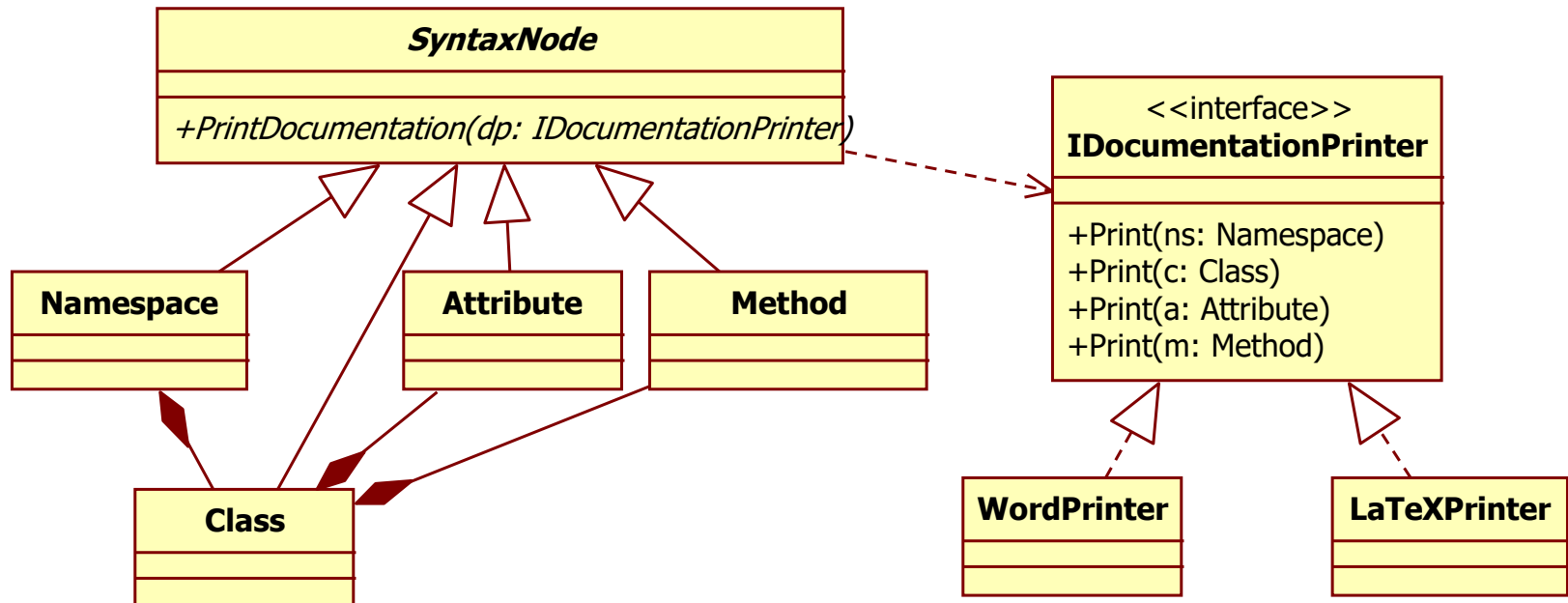
```
public void PrintDocumentation(IDocumentPrinter dp) {  
    if (this is Namespace) dp.Print((Namespace)this);  
    if (this is Class) dp.Print((Class)this);  
    // ...  
}
```

Problem: class depends on its descendants

17. 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

17. Solution



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

18. Use protected only for methods and never for attributes

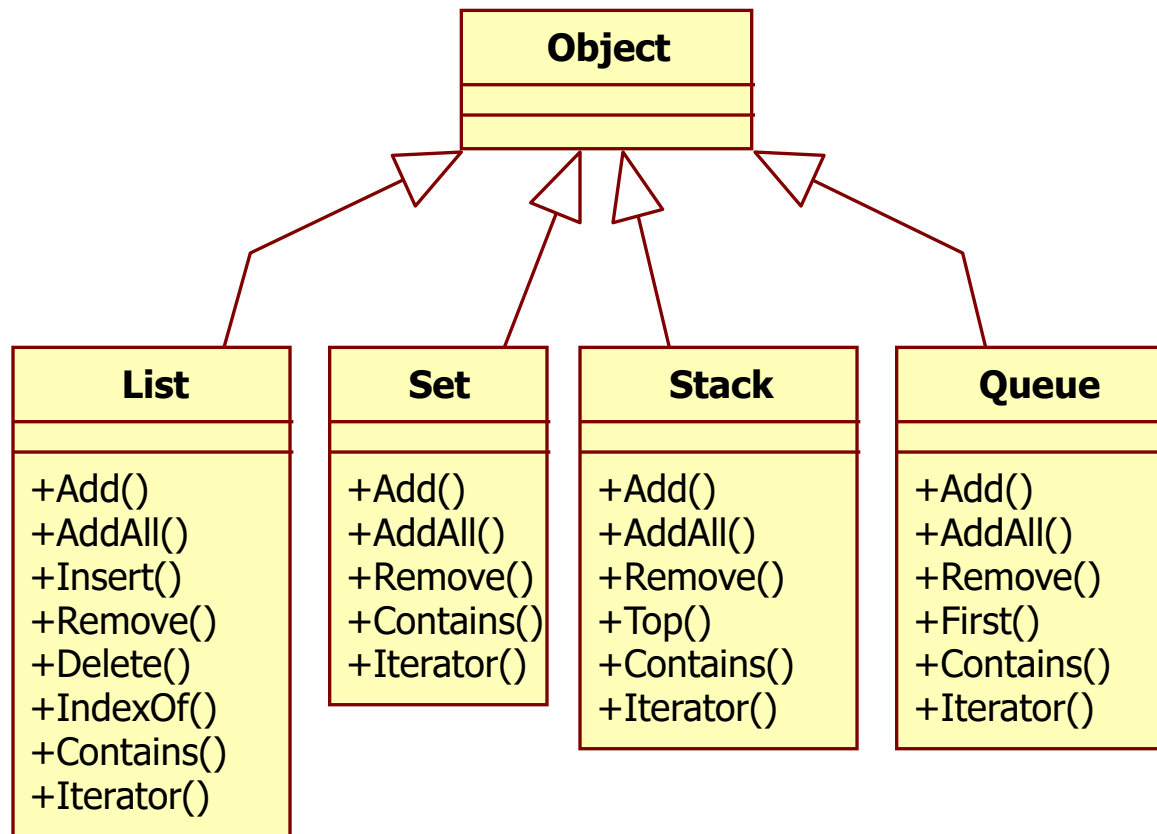
- Problems if violated:
 - descendants accessing protected attributes can bring the object into a bad state
- Rules:
 - attributes should always be private
 - if the value of attributes is needed, provide protected accessor methods for them
 - do not make these method public, if it is not necessary
 - 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()?

19. 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

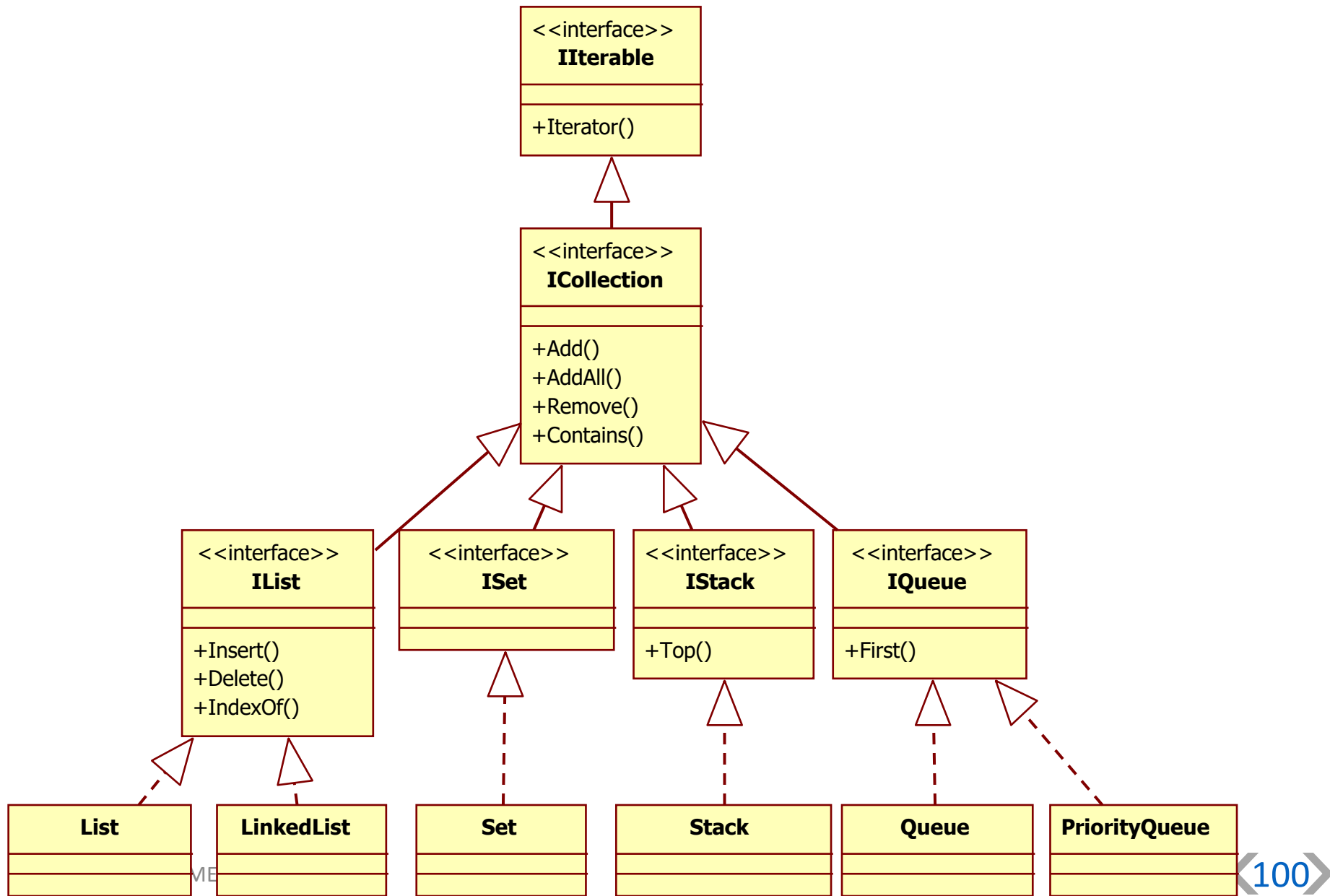
I10. Abstract classes should be at the root of the inheritance hierarchy

- 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

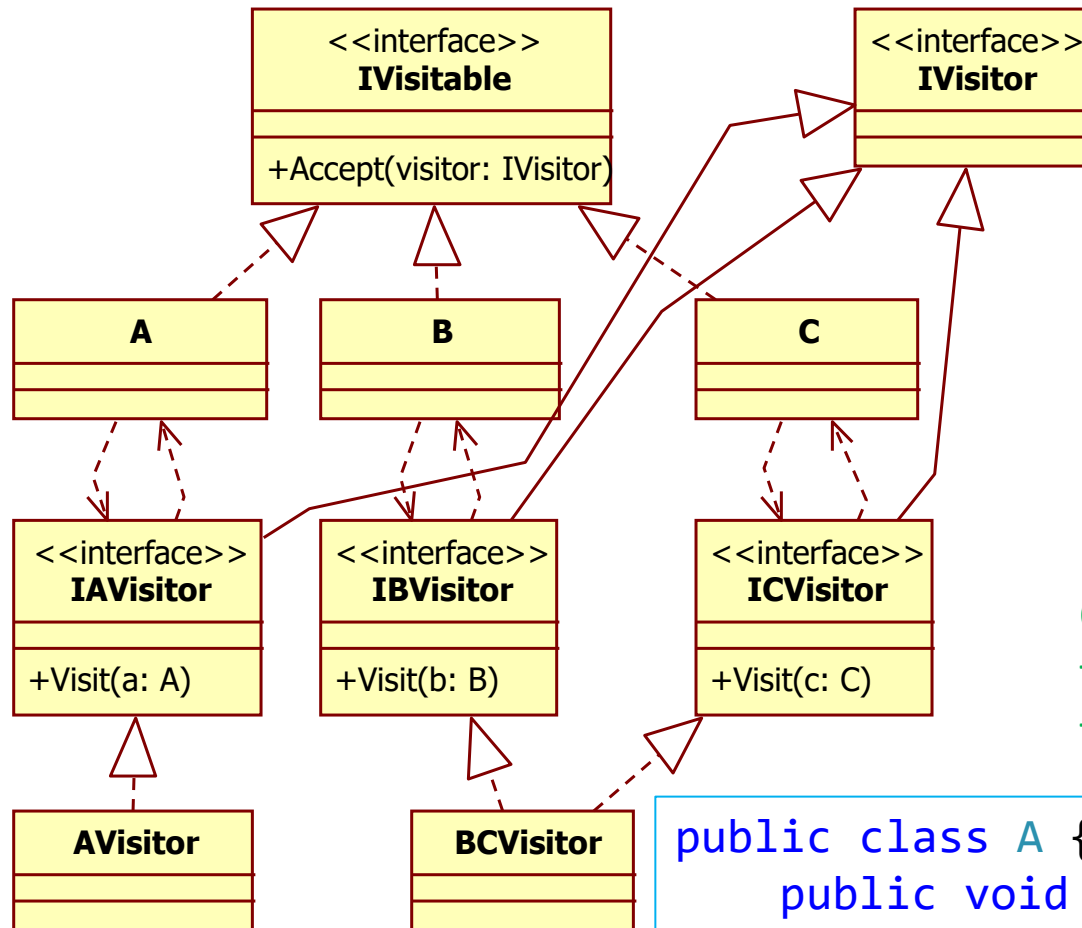
I12. Never test for the type of an object, use polymorphism instead

- 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
    }
}
// ...
```

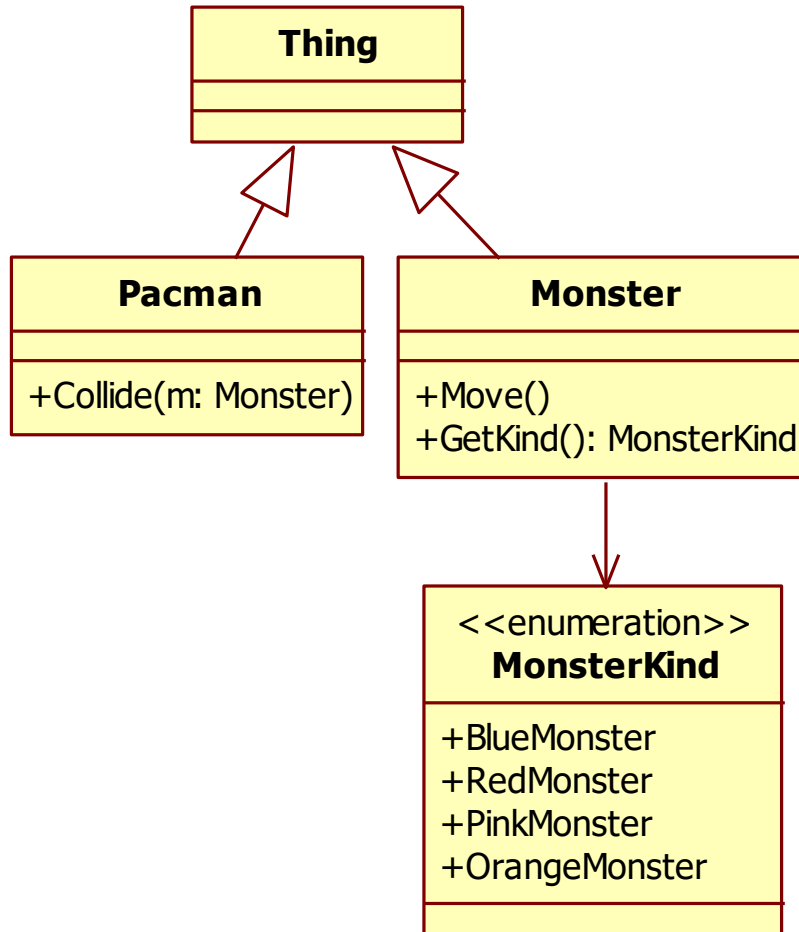
I12. Solution for a non-OCF-violating visitor: Acyclic Visitor



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;
        }
    }
}
```

Problem: dynamic type checking through a type code

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

- Problems if violated:

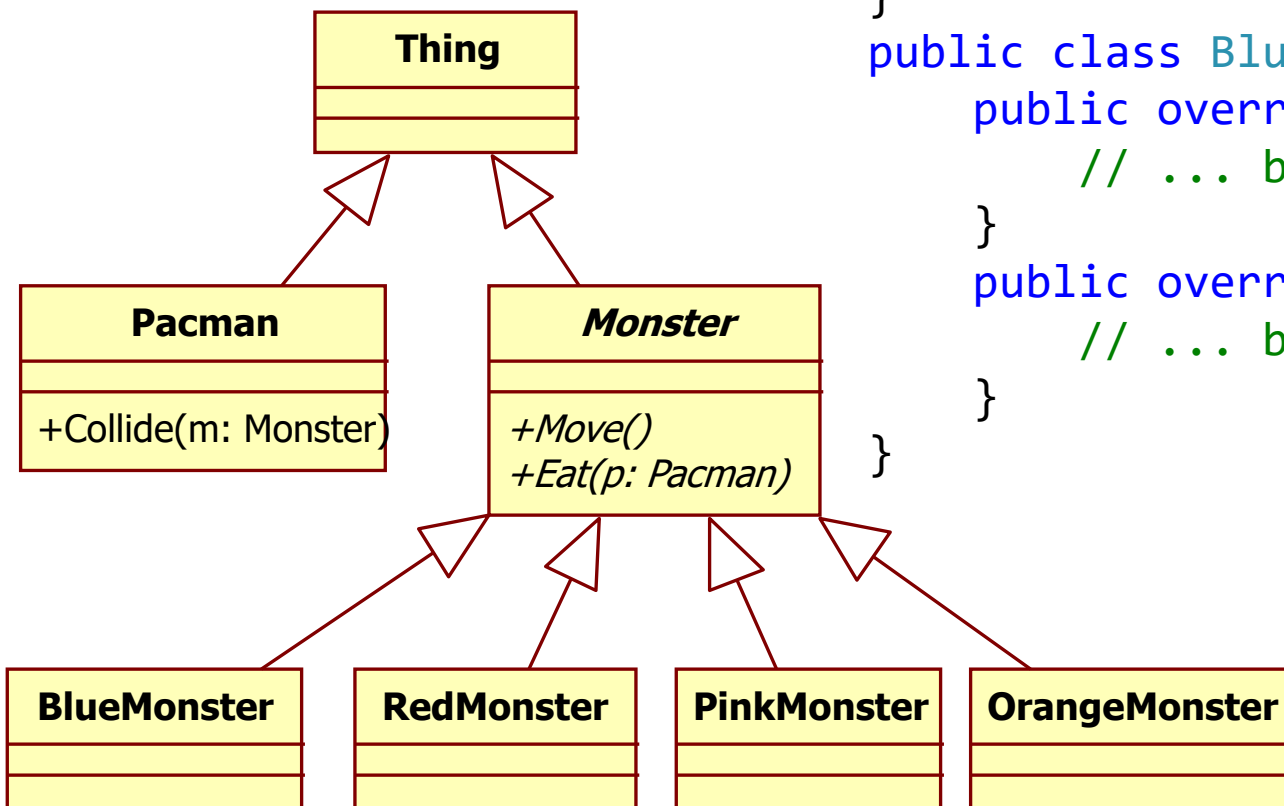
- similar to dynamic type test
- violation of LSP and OCP

- 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

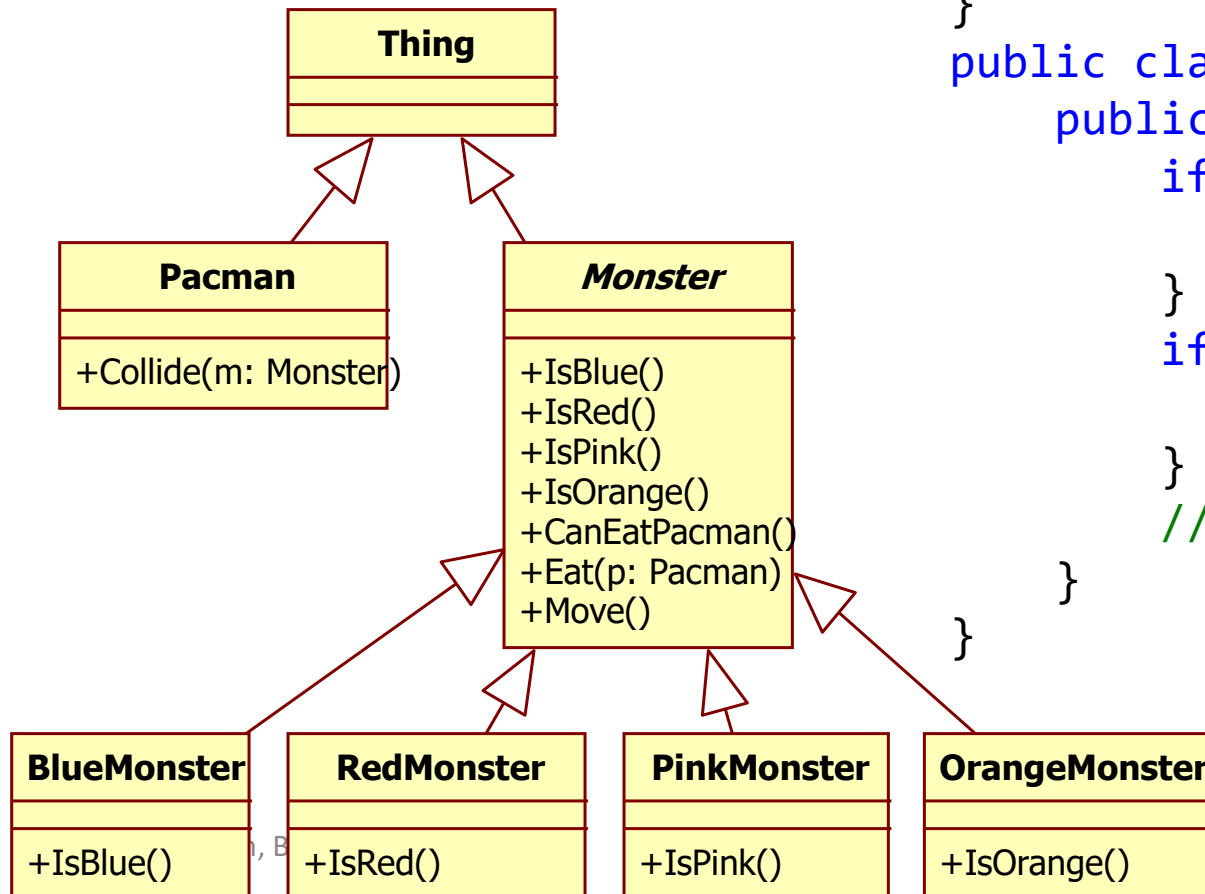
```
public class Pacman : Thing {  
    public void Collide(Monster m) {  
        m.Eat(this);  
    }  
}  
  
public class BlueMonster : Monster {  
    public override void Eat(Pacman p) {  
        // ... blue behavior  
    }  
    public override void Move() {  
        // ... blue behavior  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



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

I14. Problem

Problem: dynamic type checking
through capability methods



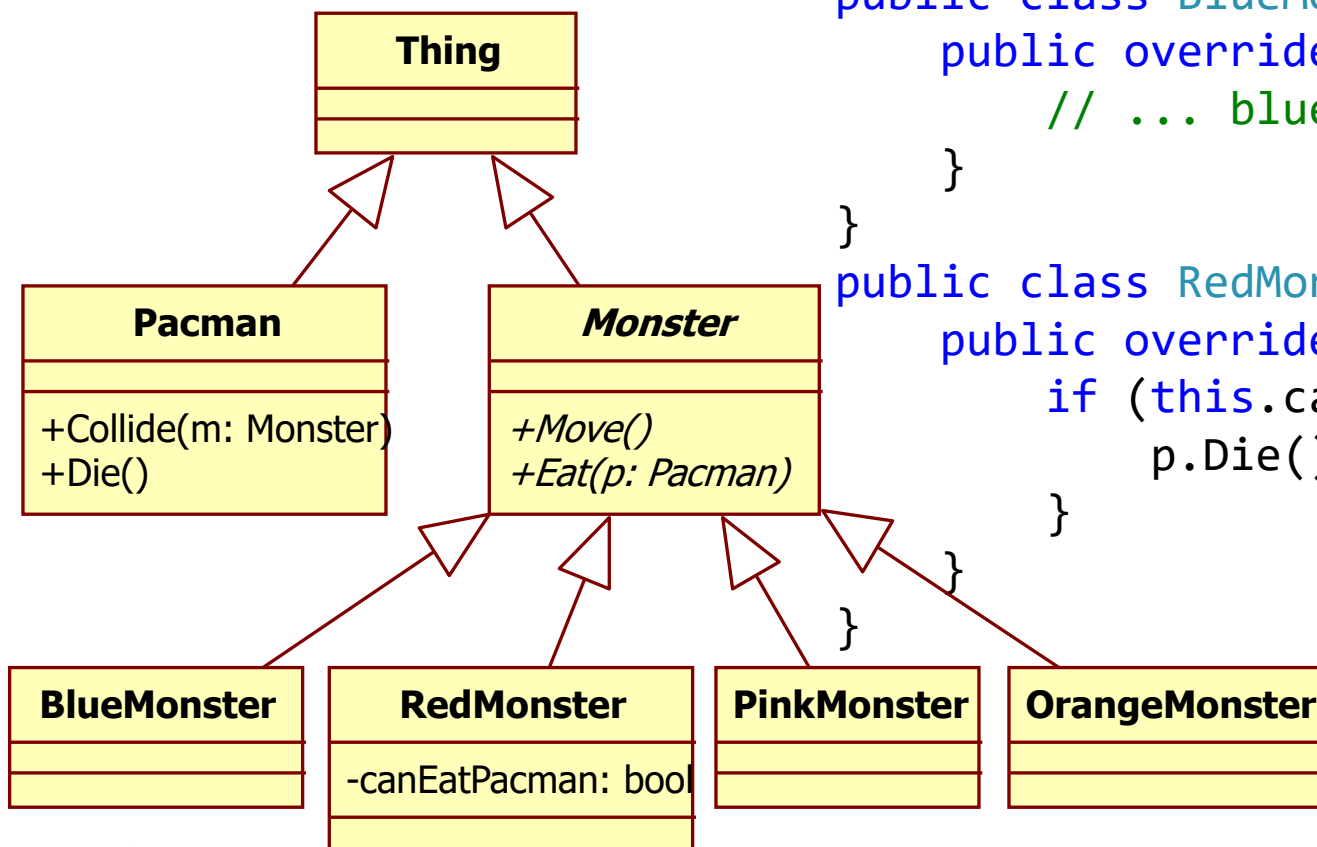
```
public class Pacman : Thing {
    public void Collide(Monster m) {
        if (m.CanEatPacman()) {
            m.Eat(this);
        }
    }
}

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

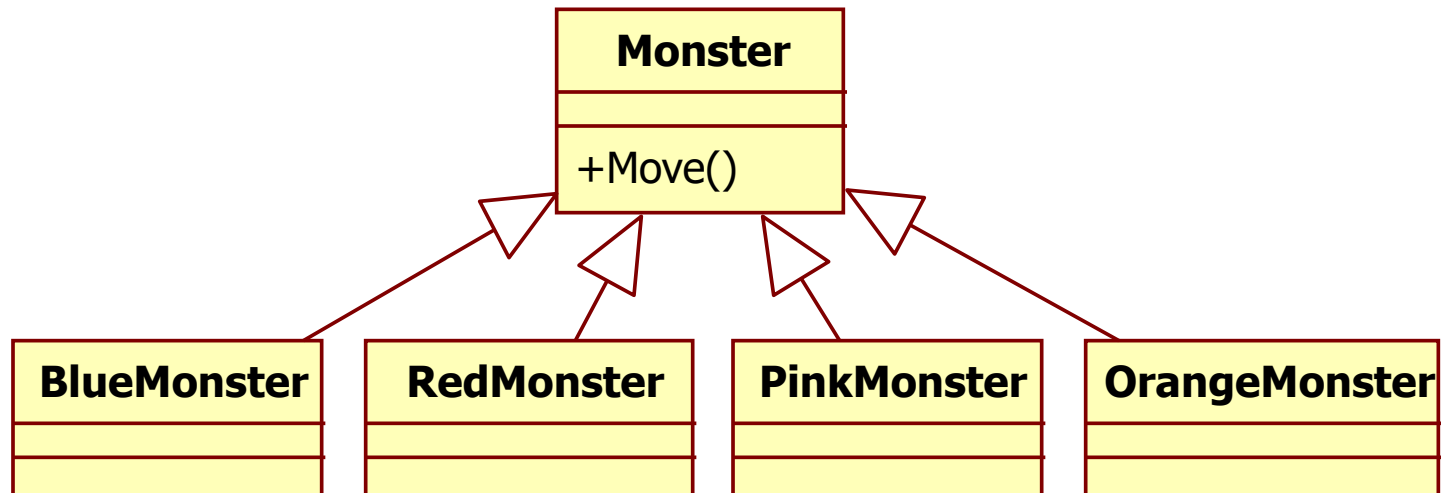


```
public class Pacman : Thing {
    public void Collide(Monster m) {
        m.Eat(this);
    }
}

public class BlueMonster : Monster {
    public override void Move() {
        // ... blue behavior
    }
}

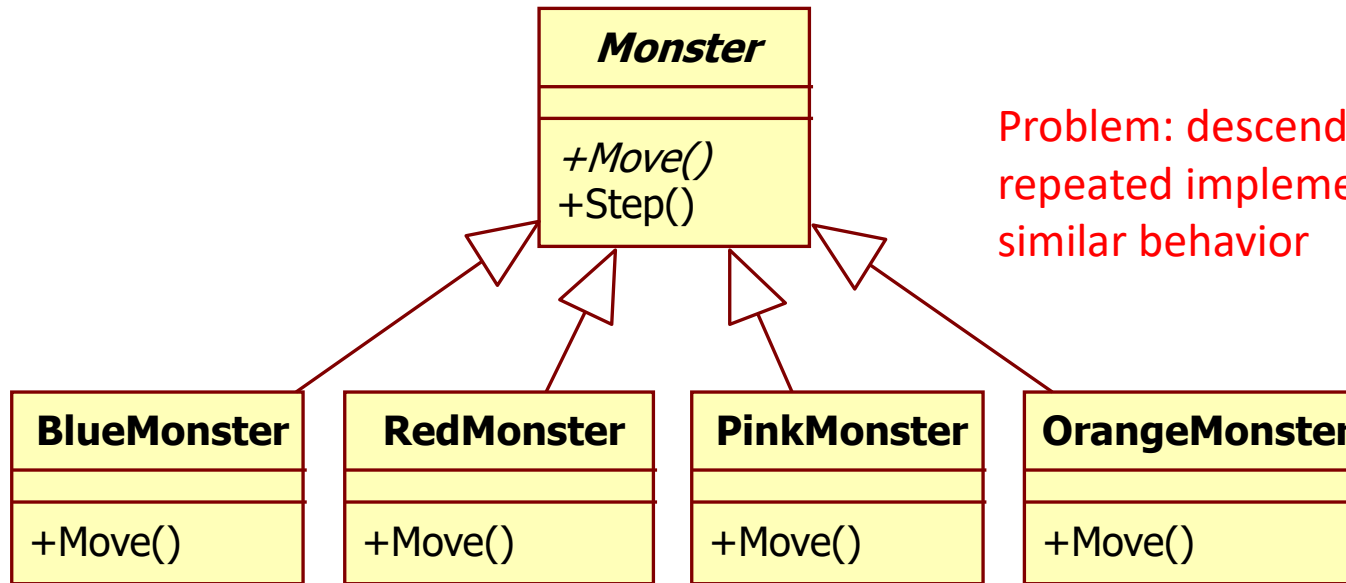
public class RedMonster : Monster {
    public override void Eat(Pacman p) {
        if (this.canEatPacman) {
            p.Die();
        }
    }
}
```

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:

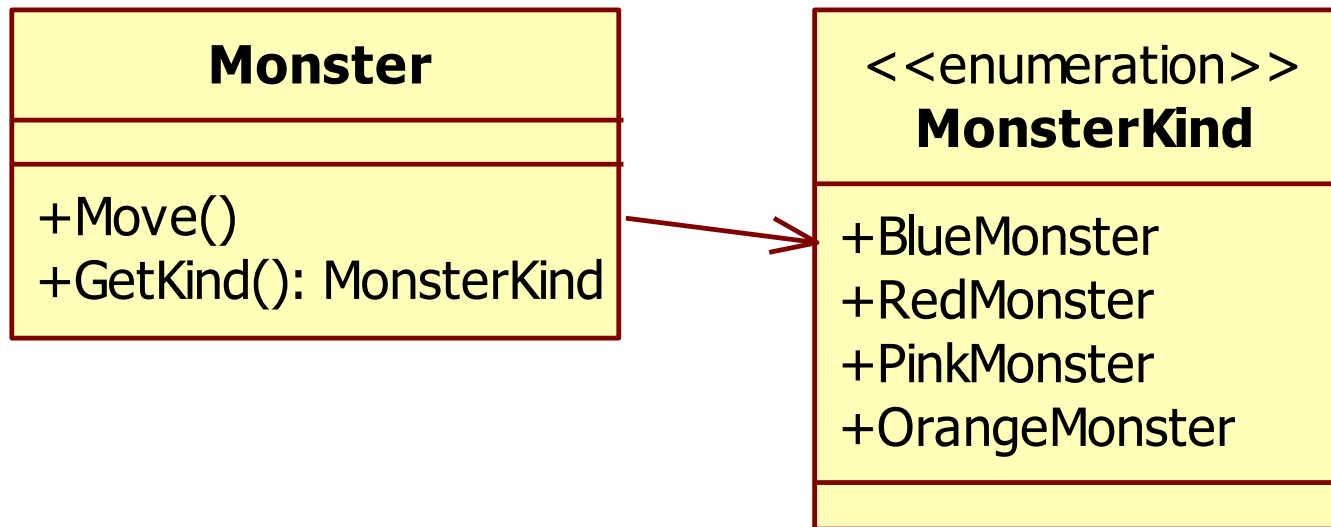
- multiple descendants with the same or similar behavior
- multiple descendants with no added behavior

■ 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)

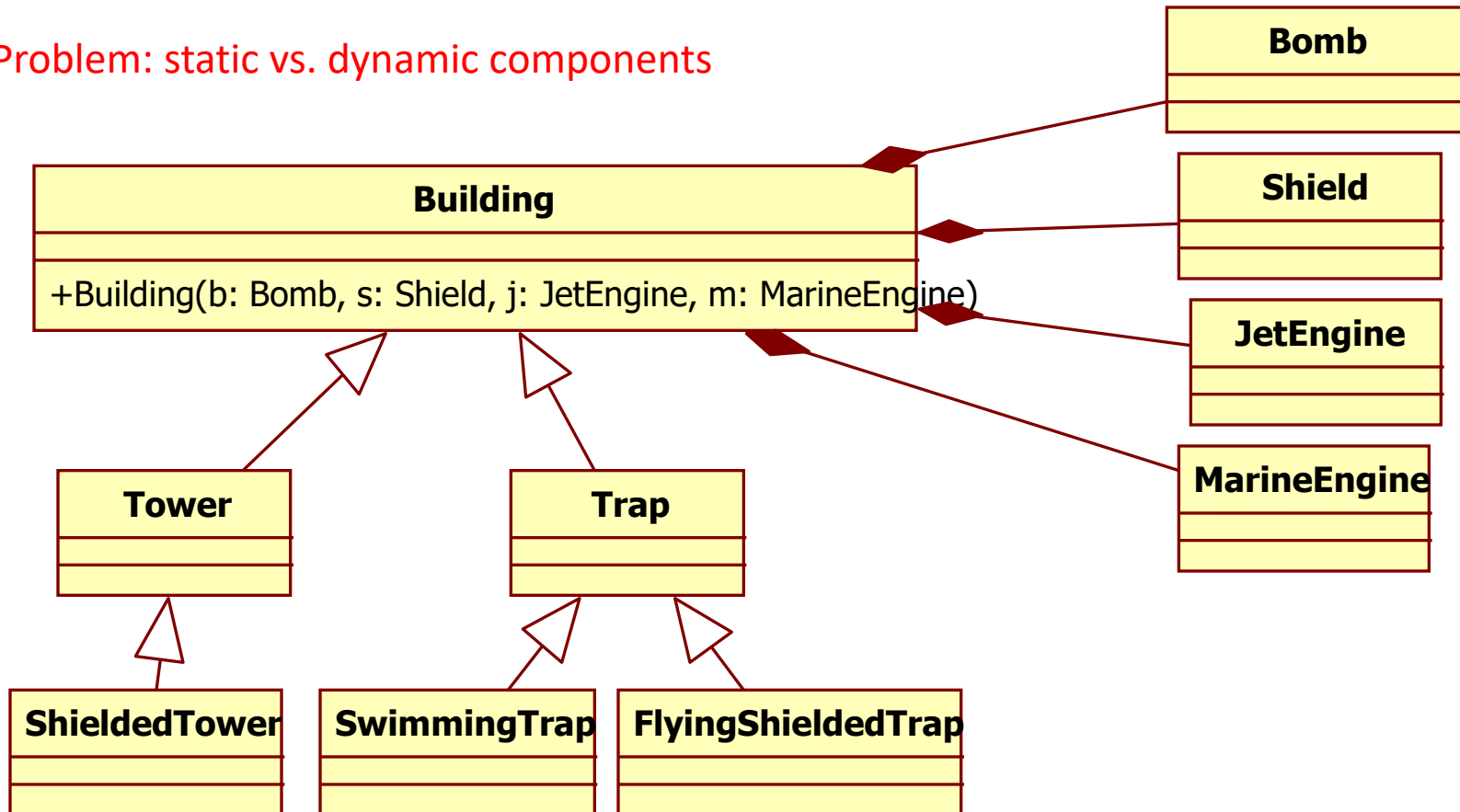
Monster
-speed: int
+Move() +Step()

```
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?

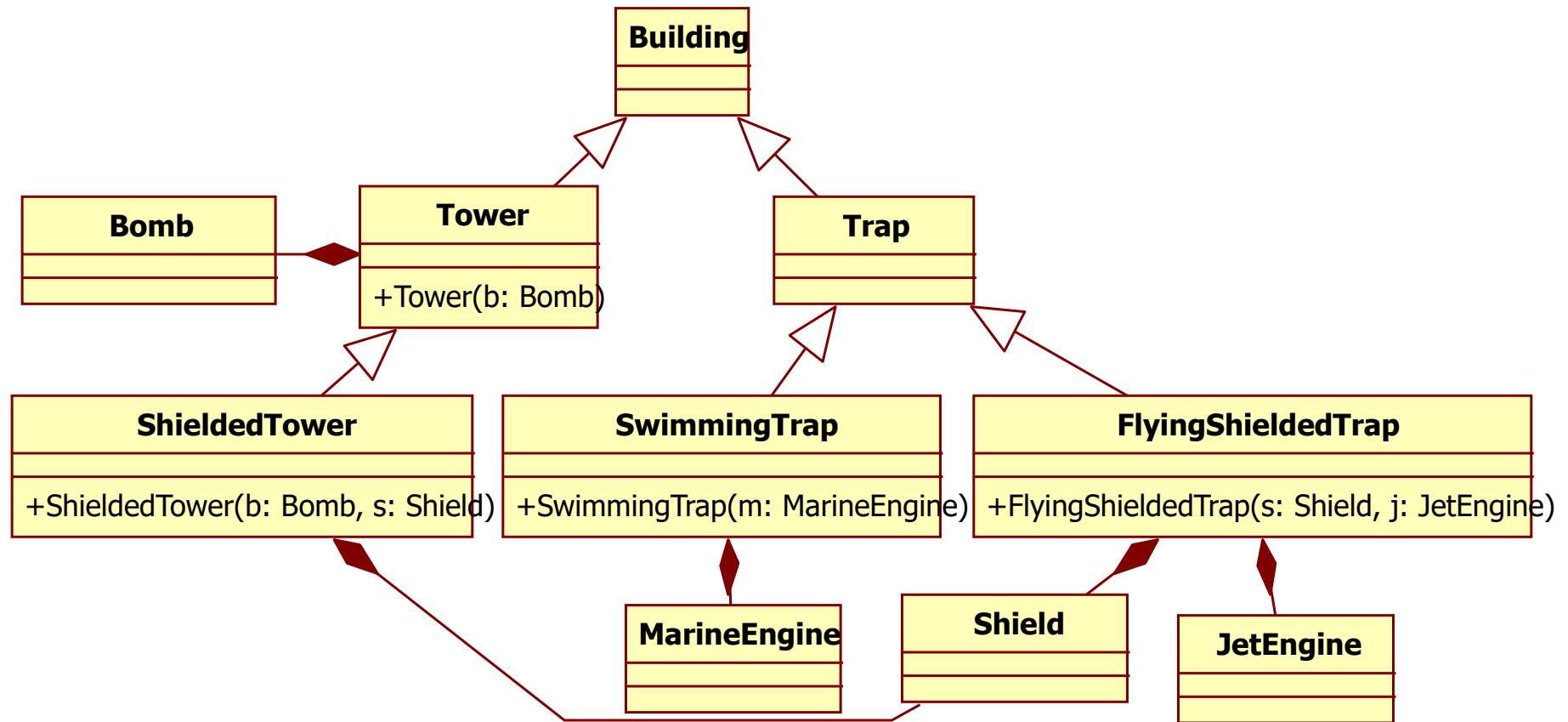
Problem: static vs. dynamic components



116. 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

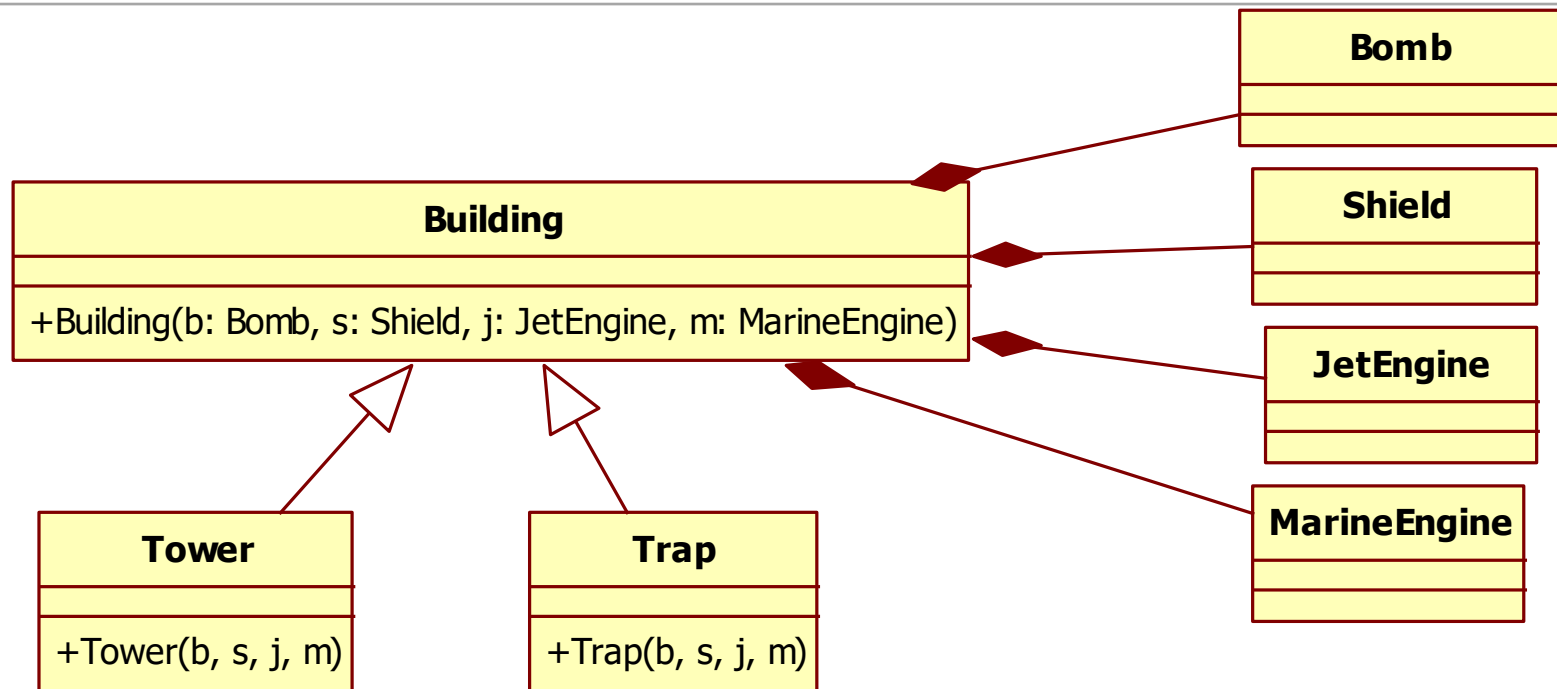
I16. Solution



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.");
    }
}

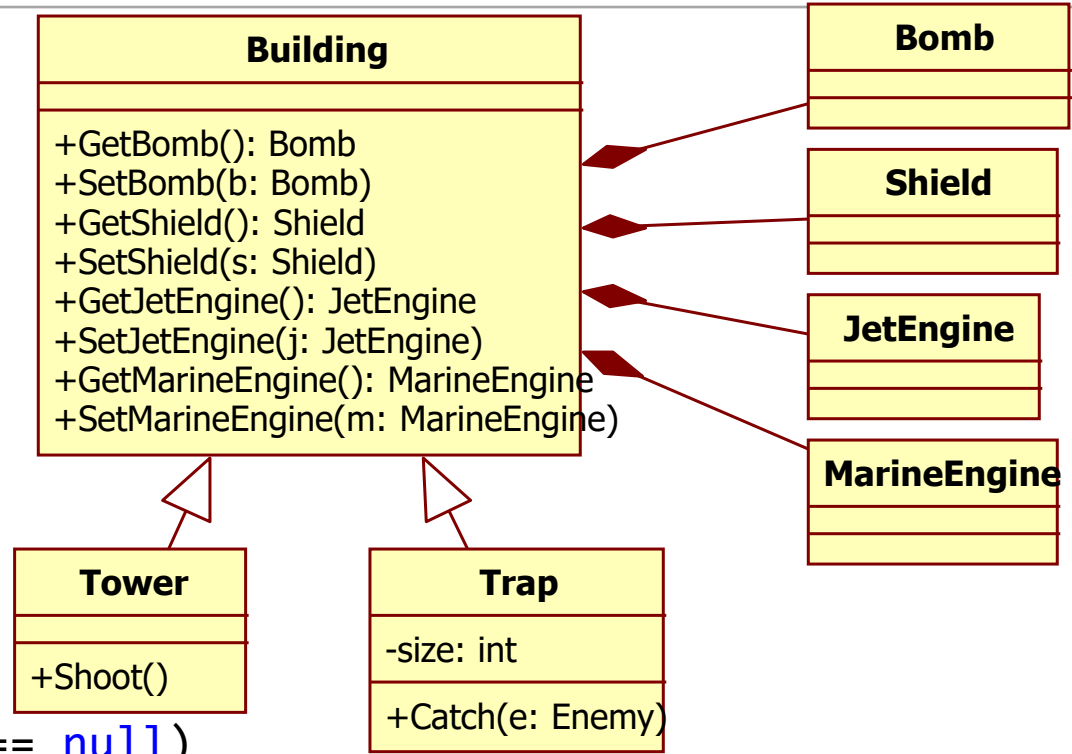
public class Tower {
    public Tower(Bomb b, Shield s, JetEngine j, MarineEngine m) {
        if (j != null) throw new Exception("Towers cannot fly.");
        if (m != null) throw new Exception("Towers cannot swim.");
    }
}
```

ázs Simon, BME, IIT

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.");
    }
}

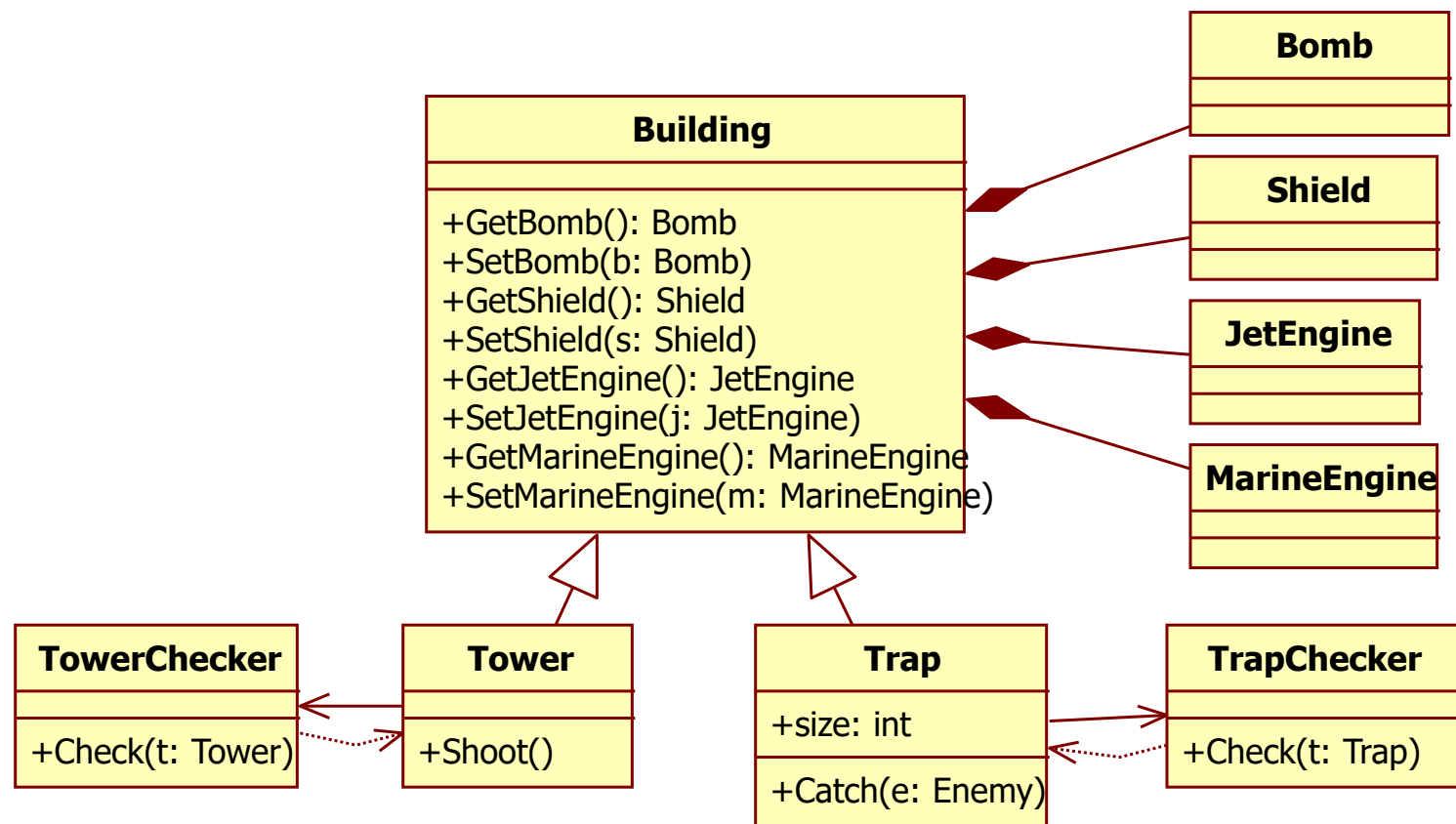
public class Trap {
    public void Catch(Enemy e) {
        if (this.size >= 5 && this.GetMarineEngine() != null)
            throw new Exception("A trap of size 5 cannot swim.");
    }
}

```

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



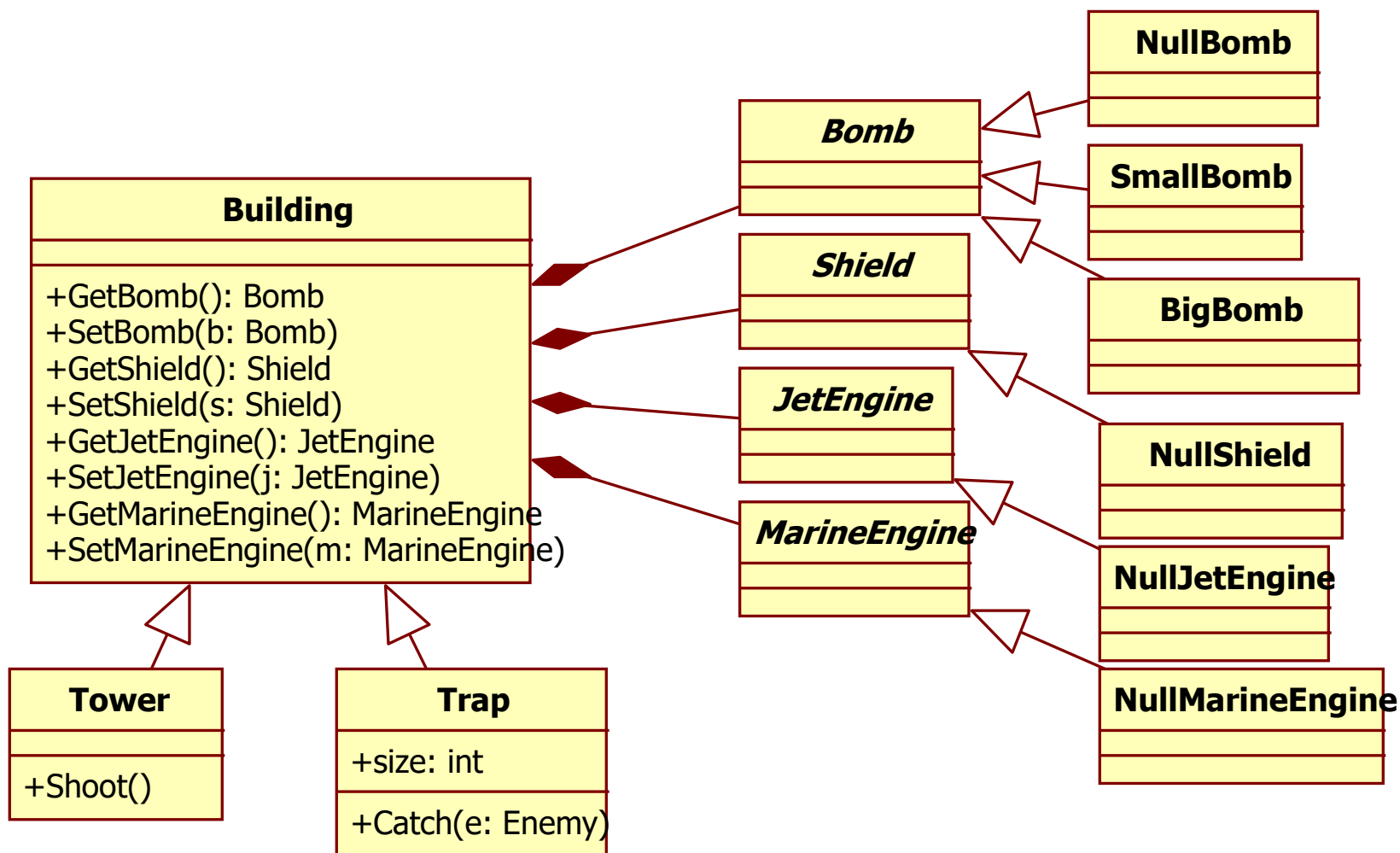
```
public class Tower {  
    public void Shoot() {  
        this.towerChecker.Check(this  
    }  
}
```

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

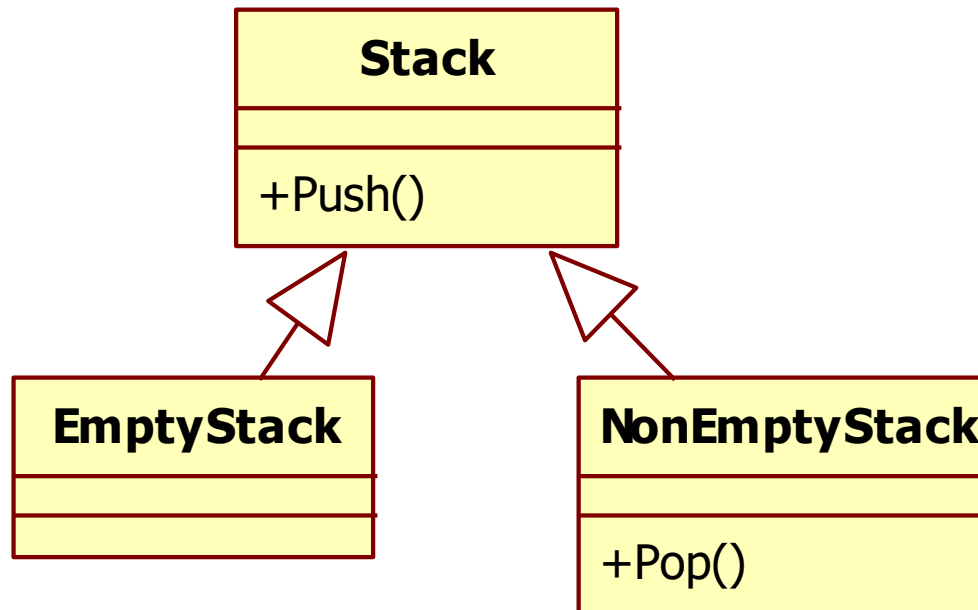
I20. Model optional elements as containment and never as inheritance

- 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

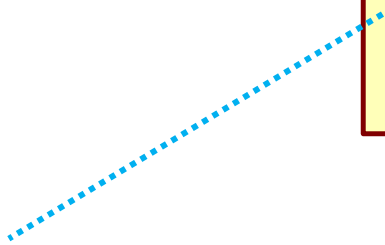
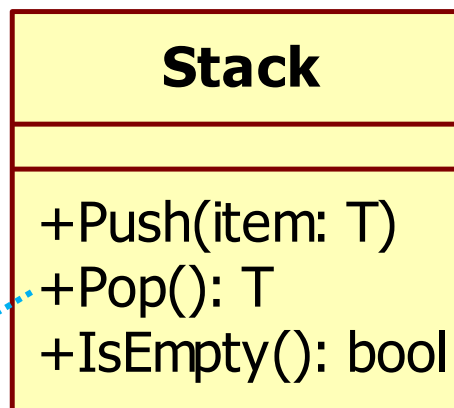


Problem: toggling of types

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

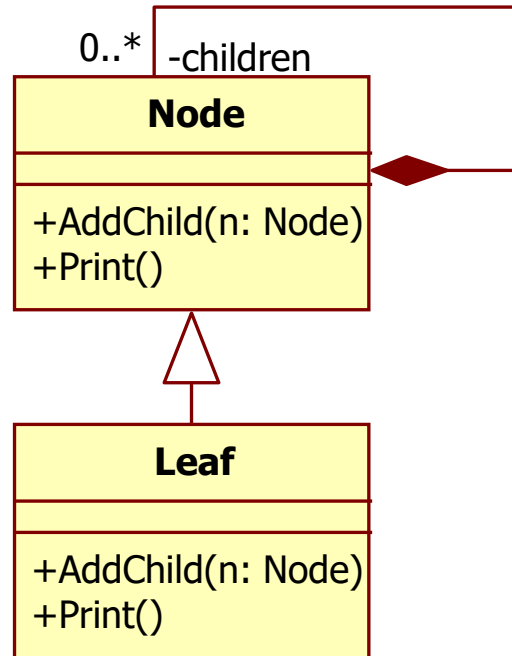


```
T Pop() {  
    if (this.IsEmpty()) throw new EmptyStackException();  
    // ...  
}
```

122. 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

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

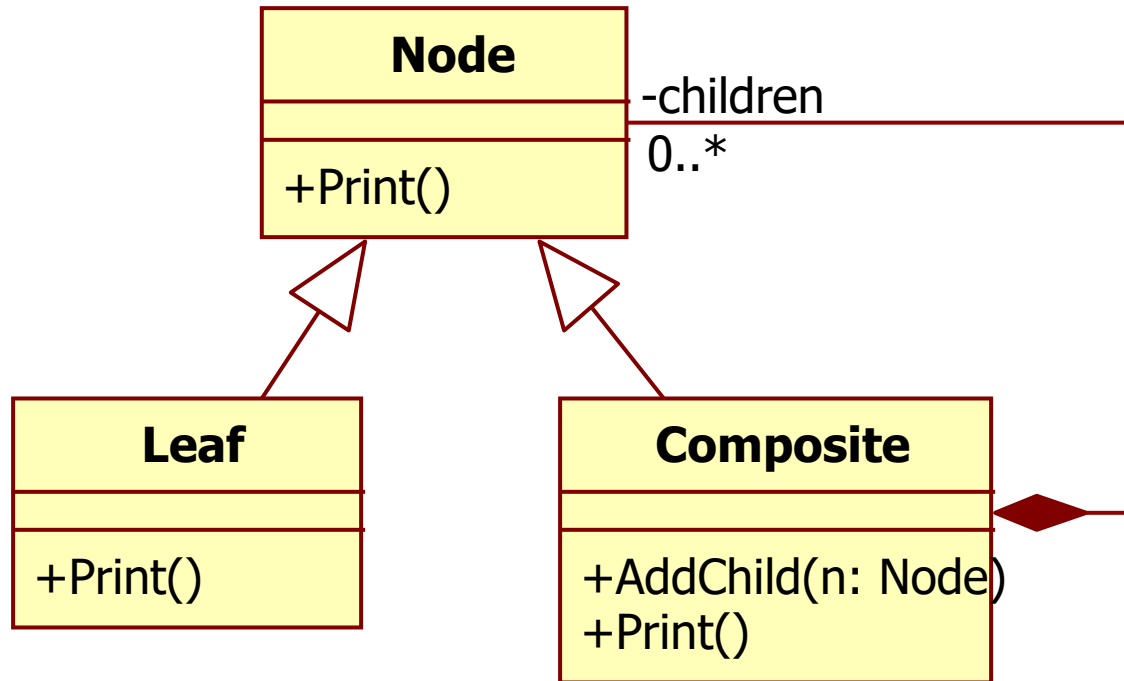
- Problems if violated:

- 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

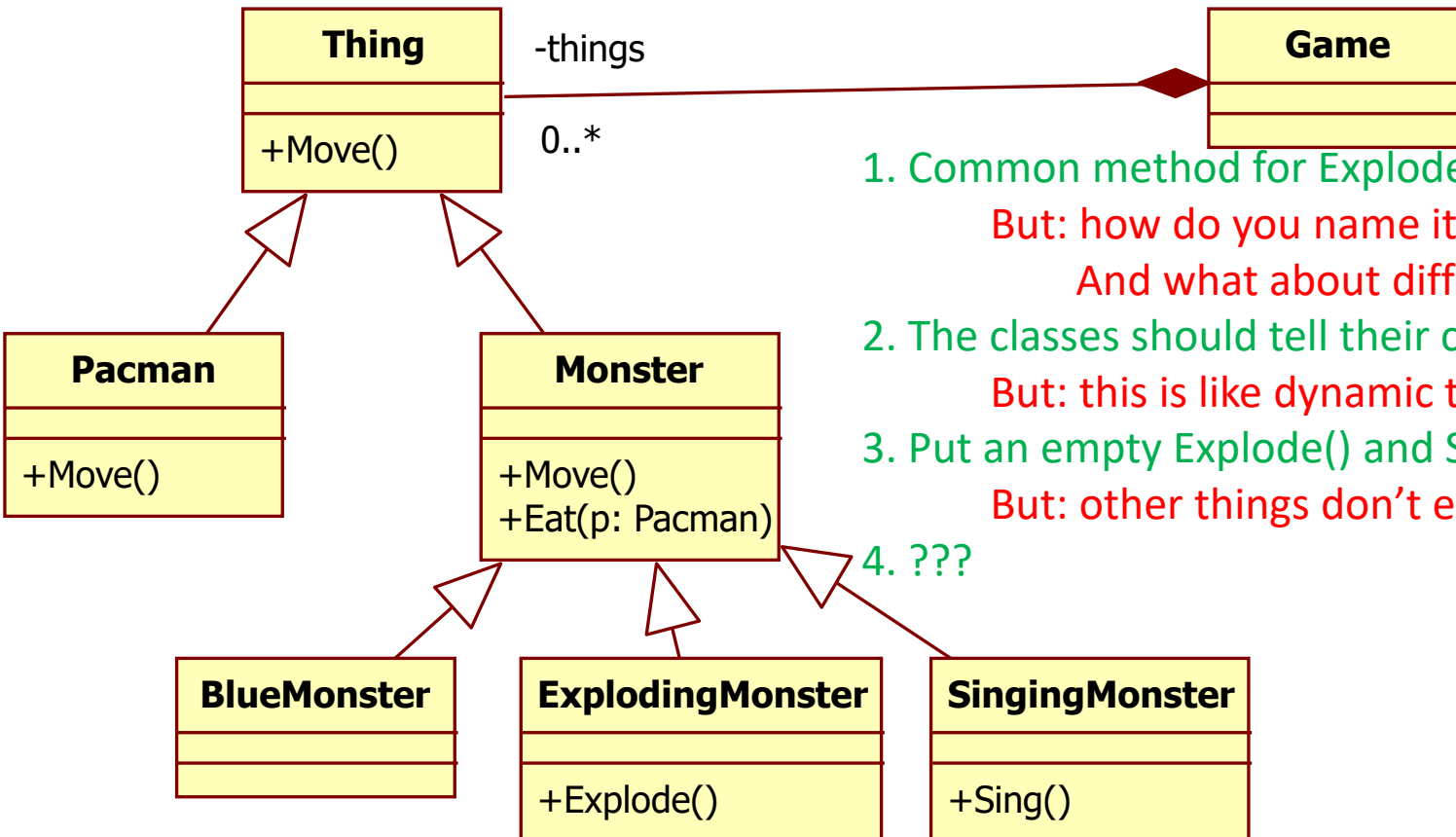
- Problems if violated:
 - modules may not be reused in other projects
 - changes in requirements may result in large changes in the design
- Rules:
 - consider designing a general solution instead of a specific one
 - try to adhere to API design guidelines for more convenient use of modules
 - but keep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

125. 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?



1. Common method for `Explode()` and `Sing()`

But: how do you name it?

And what about different parameters?

2. The classes should tell their capabilities

But: this is like dynamic type checking

3. Put an empty `Explode()` and `Sing()` into **Thing**

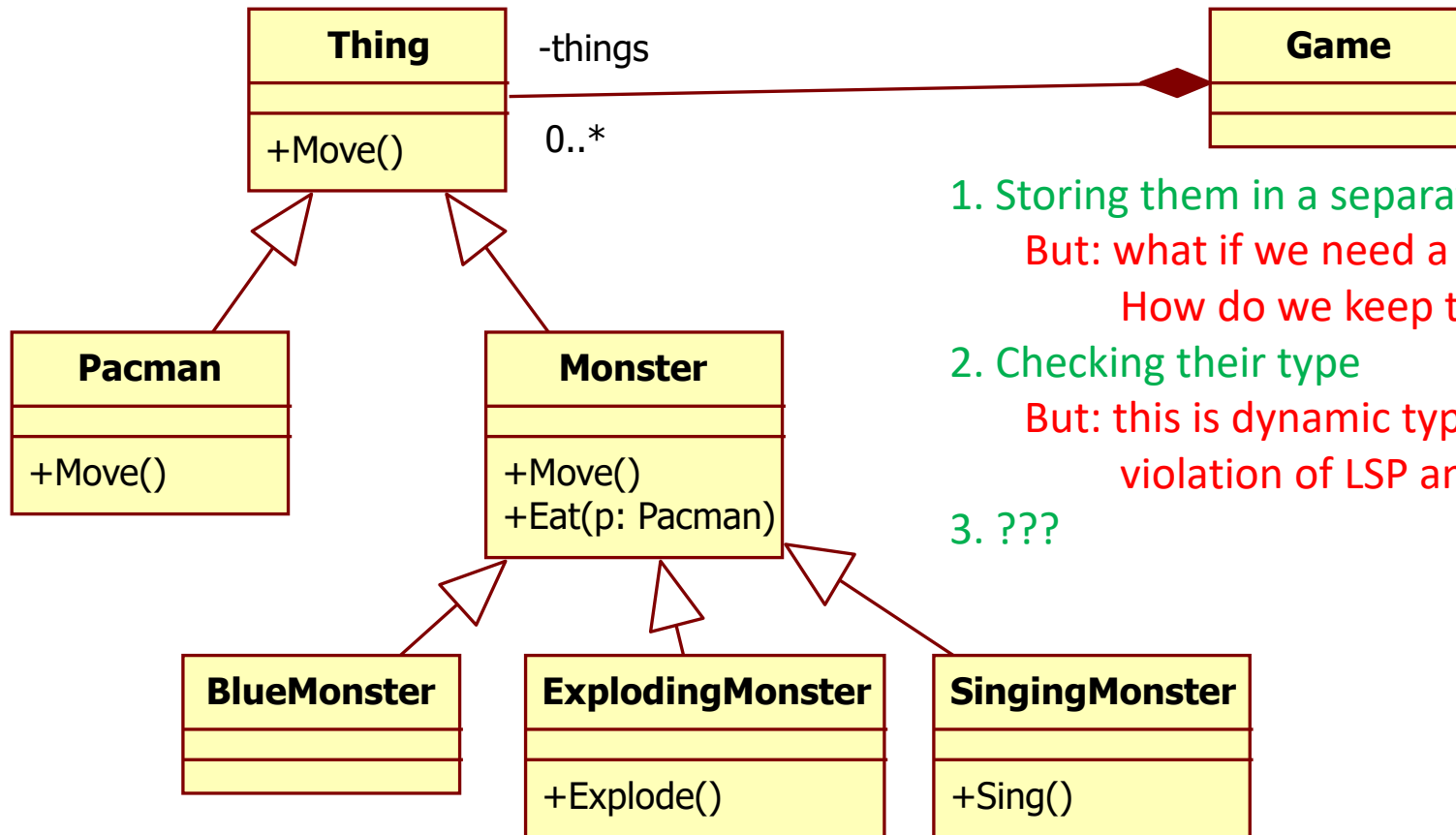
But: other things don't explode and don't sing

4. ???

None of them seems optimal

Heterogenous collection problem

How to move only the Monsters?



1. Storing them in a separate list

But: what if we need a lot of lists?

How do we keep them consistent?

2. Checking their type

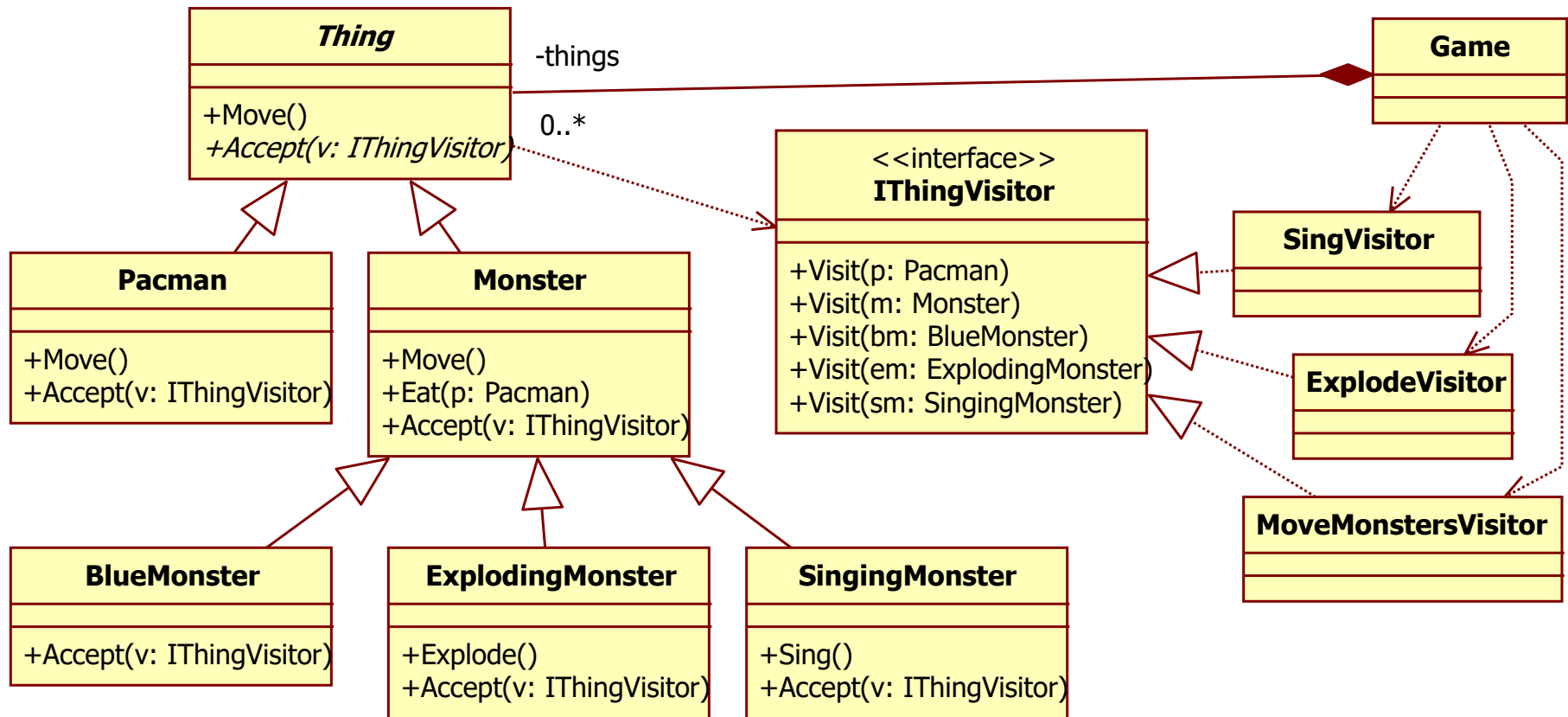
But: this is dynamic type checking,
violation of LSP and OCP

3. ???

None of them seems optimal

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