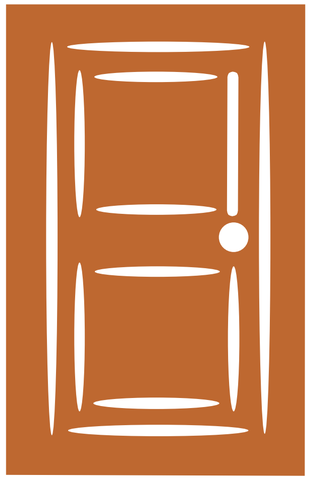
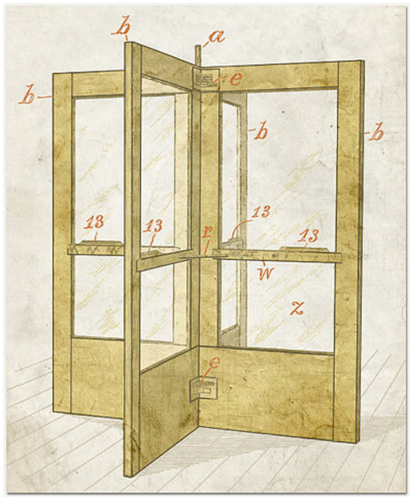
EPFL

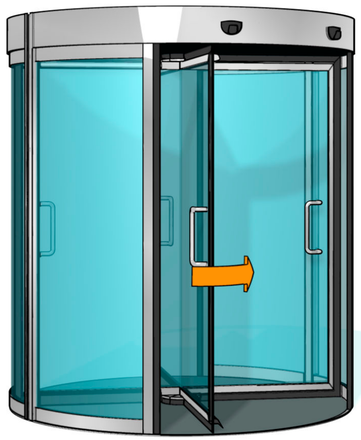
Inheritance vs. Containment

Prof. George Candea
School of Computer & Communication Sciences

Inheritance







- Inherit properties of base class
  - *e.g., door vs. specific doors*
  - *polymorphism: operations that adjust at runtime*
- Use only when it simplifies design
  - *rich set of operations on the base class*
  - *mapping to real-world inheritance*
- Containment
  - *is containment a better choice than inheritance ?*

Inheritance vs. Containment

```
class Passenger {
    FullName name;
    Address address;
    PhoneNumber number;
}


class VIP extends Passenger {
    FrequentFlyerNumber account;
}
```

- Inheritance = “is a”
  - *class is a specialization of another class*
  - *share common data and methods*
- Containment = “has a”
  - *class is implemented with the help of another*
  - *accesses are translated and forwarded*

Liskov Substitution Principle (LSP)

Barbara Liskov

Let q(x) be a property provable about objects x of type T.
Then q(y) should be provable for objects y of type S where S is a subclass of T.

⇒

- LSP intuition
  - *subclass is specialized version of base class*
  - *all methods of subclass usable through base class interface without knowing the type*
  - *base class can be replaced by a subclass, and client code will still be correct*

- *subclass must preserve superclass' invariants*
- *subclass not allowed to strengthen preconditions*
- *subclass not allowed to weaken postconditions*

Liskov Substitution Principle

```
public class Rectangle {
    private int width;
    private int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }
}
```

```
public class Square extends Rectangle {

    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    public void setHeight(int height) {
        super.setWidth(height);
        super.setHeight(height);
    }
}
```

```
void initialize(Rectangle r) {
    r.setWidth(5);
    r.setHeight(10);
    assert(r.getArea() == 50);
}
```

Liskov Substitution Principle

```
public class Rectangle {
    private int width;
    private int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }
}
```

```
public class Square extends Rectangle {

    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    public void setHeight(int height) {
        super.setWidth(height);
        super.setHeight(height);
    }
}
```

```
public class Rectangle {
    private int width;
    private int height;

    Rectangle(int width, int height) {
        this.height = height;
        this.width = width;
    }

    public int getArea() {
        return width * height;
    }
}
```

```
public class Square extends Rectangle {

    Square(int height, int width) throws IllegalArgumentException {
        super(width, height);
        if (height != width) {
            throw new IllegalArgumentException();
        }
    }
}
```

```
void initialize(Rectangle r) {
    r.setWidth(5);
    r.setHeight(10);
    assert(r.getArea() == 50);
}
```

Liskov Substitution Principle

```
public class Rectangle {
    private int width;
    private int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }
}
```

```
public class Rectangle {
    private int width;
    private int height;

    Rectangle(int width, int height) {
        this.height = height;
        this.width = width;
    }

    public int getArea() {
        return width * height;
    }
}
```

```
public class Square extends Rectangle {

    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    public void setHeight(int height) {
        super.setWidth(height);
        super.setHeight(height);
    }
}
```

```
public class Square extends Rectangle {

    Square(int height, int width) throws IllegalArgumentException {
        super(width, height);
        if (height != width) {
            throw new IllegalArgumentException();
        }
    }
}
```

```
Rectangle r = new Rectangle(5,10);
```

```
Square s = new Square(10,10);
```

```
Square s = new Square(5,10);
```

IllegalArgumentException

Design for Inheritance

Modifier	Classes or subclasses in package	Subclasses outside package	Classes outside package
`public`	Yes	Yes	Yes
`protected`	Yes	Yes	No
no modifier	Yes	No	No
`private`	No	No	No

- Document overriding, or prohibit it
  - *use* `final` *for methods you don't want overridden*
  - *remember access rules for attributes and methods*
  - *use* `final` *class to prevent it being subclassed and force composition instead*

Inheritance and Encapsulation

```
public class CharacterSet {
    protected StringBuffer s;
    // invariant: s != null
    ...
}
```

```
public class PersistentCharacterSet extends CharacterSet {

    public loadFromFile(String name) {
        try {
            // load contents ...
        } catch (IOException e) {
            s = null;
        }
    }
}
```

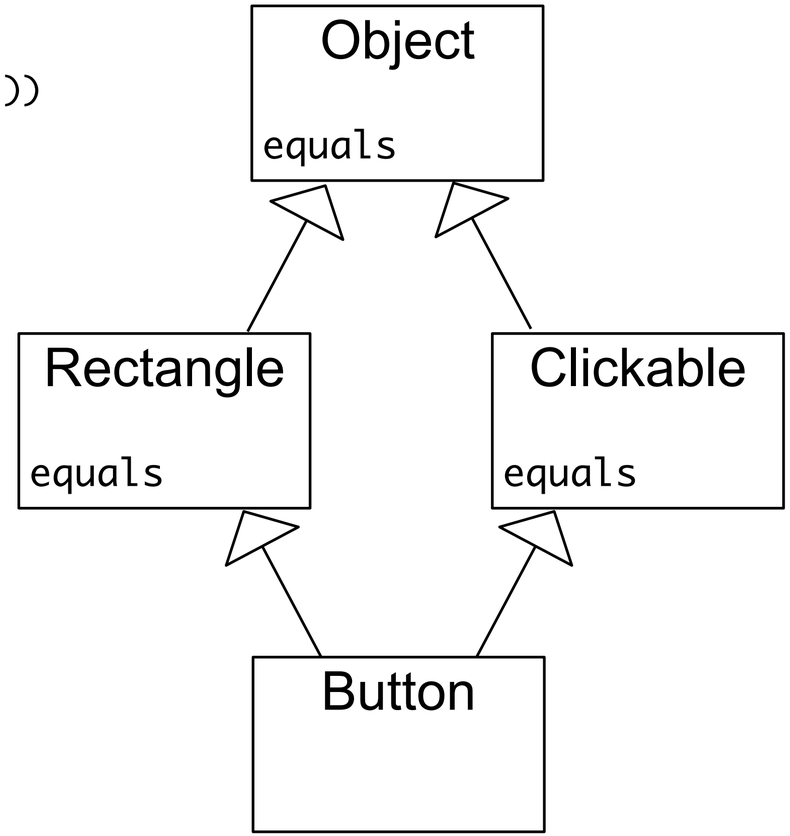
- Inheritance breaks encapsulation
  - *in turn, this increases complexity and coupling*
- Favor `private` over `protected`
  - *prevent subclass from violating invariants*
  - *if data access is necessary, use* `protected` *getter/setter methods that can protect invariants*

Inheritance Hierarchies

- Avoid deep hierarchies
  - *max 3 levels of inheritance, max 7±2 subclasses*
  - *deep inheritance trees produce higher bug rates*
- Avoid linear hierarchies
  - *single derived class = warning sign for mistaken "designing ahead"*
  - *it's better to design easy-to-change classes, and refactor later if needed*
- Push common interfaces, data, and behavior as high up as possible

Multiple Inheritance

```
class Button extends Rectangle, Clickable {
    // ...
}

// ...
if (button.equals(obj))
// ...
```



- Hardly ever a good reason to do it
  - *even if your language allows it, avoid multiple inheritance*
- Example problem
  - *the “Diamond Problem”*

Interface Inheritance and Mixins

```
class Button extends Rectangle, Clickable {
    // ...
}
```

```
public class JButton extends AbstractButton
implements Accessible, ImageObserver, ItemSelectable, MenuContainer, Serializable, SwingConstants {
    // ...
}
```

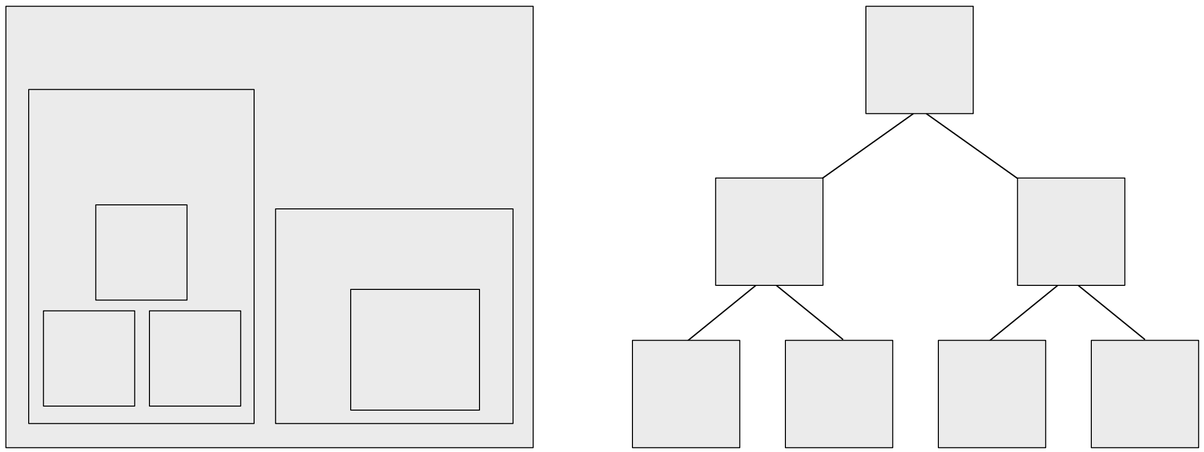
- Multiply inherited interfaces
  - *e.g., in Java, C#*
  - *only abstract methods, no implementations or fields*
- Mixins
  - *e.g., in Scala, Python, Perl, Ruby*

Interface Inheritance and Mixins

```
abstract class AbsIterator {
  type T
  def hasNext: Boolean
  def next: T
}

trait RichIterator extends AbsIterator {
  def foreach(f: T => Unit) { while (hasNext) f(next) }
}

class StringIterator(s: String) extends AbsIterator {
  type T = Char
  private var i = 0
  def hasNext = i < s.length()
  def next = { val ch = s charAt i; i += 1; ch }
}

object StringIteratorTest {
  def main(args: Array[String]) {
    class Iter extends StringIterator(args(0)) with RichIterator
    val iter = new Iter
    iter foreach println
  }
}
```

- Multiply inherited interfaces
  - *e.g., in Java, C#*
  - *only abstract methods, no implementations or fields*
- Mixins
  - *e.g., in Scala, Python, Perl, Ruby*
  - *inheritance of implementations from multiple mixins is allowed*
  - *orthogonal functionality, single purpose, not instantiable on their own*

Containment vs. Inheritance



- Containment
  - *use when classes share common data but not behavior*
  - *containing class controls the interface*
  - *could lead to excessive method forwarding*
- Inheritance
  - *use if multiple classes share common behavior*
  - *avoid if it violates the Liskov Substitution Principle*
  - *only inherit what is truly shared (not necessarily data)*
  - *base class controls interface and provides implementation*