The SOLID Principles

Klaus Iglberger, CppEurope, Online Edition

klaus.iglberger@gmx.de

C++ Trainer since 2016

Author of the blaze C++ math library

(Co-)Organizer of the Munich C++ user group

Regular presenter at C++ conferences

Email: klaus.iglberger@gmx.de

Klaus Iglberger

Software

Software

Soft

=

Easy to change and extend

”Dependency is the key problem in software development at all scales.”
(Kent Beck, TDD by Example)

The SOLID Principles

Single-Responsibility Principle

Open-Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

The SOLID Principles

Single-Responsibility Principle

Open-Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Robert C. Martin

Michael Feathers

SOLID

From Wikipedia, the free encyclopedia
(Redirected from SOLID (object-oriented design))

This article is about the SOLID principles of object-oriented programming. For the fundamental state of matter, see Solid. For other uses, see Solid (disambiguation).

In object-oriented computer programming, **SOLID** is a mnemonic acronym for five design principles intended to make software designs more understandable, flexible and maintainable. It is not related to the GRASP software design principles. The principles are a subset of many principles promoted by American software engineer and instructor Robert C. Martin.[1][2][3] Though they apply to any object-oriented design, the SOLID principles can also form a core philosophy for methodologies such as agile development or adaptive software development.[3] The theory of SOLID principles was introduced by Martin in his 2000 paper *Design Principles and Design Patterns*,[2][4] although the SOLID acronym was introduced later by Michael Feathers.[5]

SOLID

Principles
Single responsibility
Open–closed
Liskov substitution
Interface segregation
Dependency inversion

V • T • E

Concepts [edit]

Single-responsibility principle[6]

A class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.

Open–closed principle[7]

"Software entities ... should be open for extension, but closed for modification."

Liskov substitution principle[8]

"Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." See also design by contract.

Interface segregation principle[9]

"Many client-specific interfaces are better than one general-purpose interface."[4]

Dependency inversion principle[10]

One should "depend upon abstractions, [not] concretions."[4]

See also [edit]

- Code reuse
- Inheritance (object-oriented programming)
- Package principles
- Don't repeat yourself
- GRASP (object-oriented design)

The SOLID Principles

I will introduce the SOLID principles …

- … as guidelines not limited to OO programming
- … as general set of guidelines

The Single-Responsibility Principle (SRP)

The Single-Responsibility Principle (SRP)

”The single responsibility principle states that every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class, module or function. All its services should be narrowly aligned with that responsibility.”

(Wikipedia)

The Single-Responsibility Principle (SRP)

”Everything should do just one thing.”

(Common knowledge?)

The Single-Responsibility Principle (SRP)

”We want to design components that are self-contained: independent, and with a single, well-defined purpose ([...] cohesion). When components are isolated from one another, you know that you can change one without having to worry about the rest.”

(Andrew Hunt, David Thomas, The Pragmatic Programmer)

The Single-Responsibility Principle (SRP)

”Cohesion is a measure of the strength of association of the elements inside a module. A highly cohesive module is a collection of statements and data items that should be treated as a whole because they are so closely related. Any attempt to divide them up would only result in increased coupling and decreased readability.”

(Tom DeMarco, Structured Analysis and System Specification)

The Single-Responsibility Principle (SRP)

”A class should have only one reason to change.”
(Robert C. Martin, Agile Software Development)

The Single-Responsibility Principle (SRP)

```
class Circle
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), …

   void translate( Vector3D const& );
   void rotate( Quaternion const& );

   void draw( Screen& s, /*...*/ );
   void draw( Printer& p, /*...*/ );
   void serialize( ByteStream& bs, /*...*/ );
   // ...

 private:
   double radius;
   // ... Remaining data members
};
```

The Single-Responsibility Principle (SRP)

```
class Circle
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), …

   void translate( Vector3D const& );
   void rotate( Quaternion const& );

   void draw( Screen& s, /*...*/ );
   void draw( Printer& p, /*...*/ );
   void serialize( ByteStream& bs, /*...*/ );
   // ...

 private:
   double radius;
   // ... Remaining data members
};
```

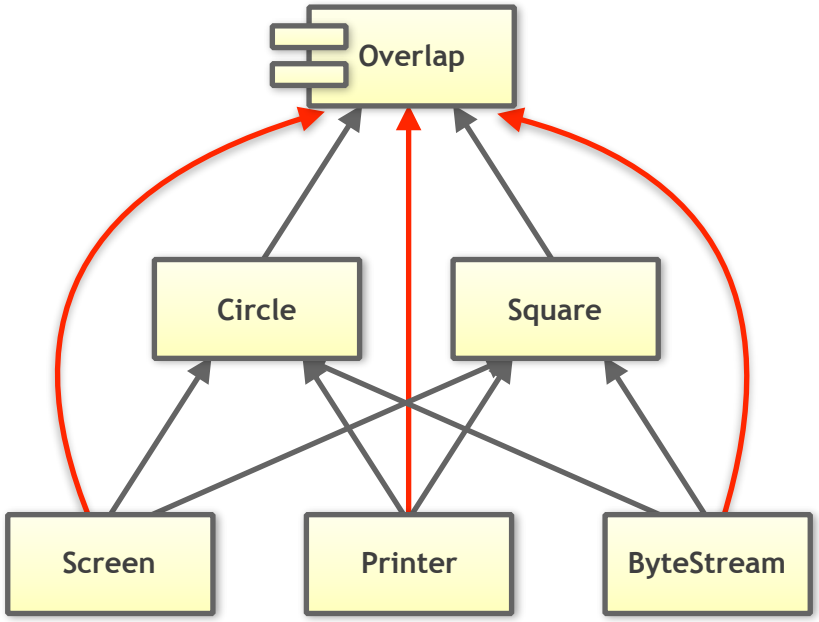
The Single-Responsibility Principle (SRP)

```
class Circle
{
 public:
   // ...
   void draw( Screen& s, /*...*/ );
   void draw( Printer& p, /*...*/ );
   void serialize( ByteStream& bs, /*...*/ );
   // ...
};
```

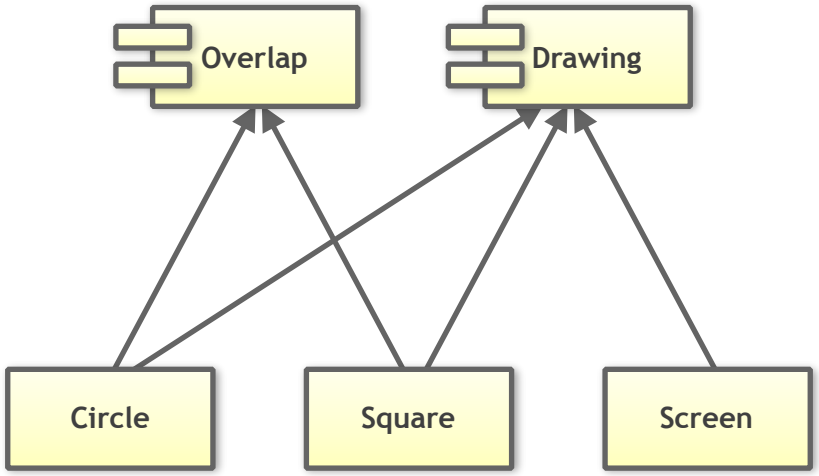
A **`Circle`** changes if …

- … the basic properties of a circle change;
- … the **`Screen`** changes;
- … the **`Printer`** changes;
- … the **`ByteStream`** changes;
- … the implementation details of **`draw()`** change;
- … the implementation details of **`serialize()`** change;
- …

The Single-Responsibility Principle (SRP)



The Single-Responsibility Principle (SRP)



The Single-Responsibility Principle (SRP)

```
namespace std {

template< typename InputIt, typename OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt dest )
{
   for( ; first != last; ++first, ++dest ) {
      *dest = *first;
   }

   return dest;
}

} // namespace std
```

The **`copy()`** function does not change. It …

- … builds on the fundamental conventions of language;
- … does not deal with memory allocation.

The Single-Responsibility Principle (SRP)

Guideline: Prefer cohesive software entities. Everything that does not strictly belong together, should be separated.

The Open-Closed Principle (OCP)

The Open-Closed Principle (OCP)

”Software artifacts (classes, modules, functions, etc.) should be open for extension, but closed for modification.”

(Bertrand Meyer, Object-Oriented Software Construction)

OCP: A Procedural Approach

```
enum ShapeType
{
   circle,
   square
};

class Shape
{
 public:
   explicit Shape( ShapeType t )
      : type{ t }
   {}

   virtual ~Shape() = default;
   ShapeType getType() const noexcept;

 private:
   ShapeType type;
};

class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : Shape{ circle }
      , radius{ rad }
      , // ... Remaining data members
   {}
```

OCP: A Procedural Approach

```
enum ShapeType
{
   circle,
   square
};

class Shape
{
 public:
   explicit Shape( ShapeType t )
      : type{ t }
   {}

   virtual ~Shape() = default;
   ShapeType getType() const noexcept;

 private:
   ShapeType type;
};

class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : Shape{ circle }
      , radius{ rad }
      , // ... Remaining data members
   {}
```

OCP: A Procedural Approach

```
enum ShapeType
{
   circle,
   square
};

class Shape
{
 public:
   explicit Shape( ShapeType t )
      , type{ t }
   {}

   virtual ~Shape() = default;
   ShapeType getType() const noexcept;

 private:
   ShapeType type;
};

class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : Shape{ circle }
      , radius{ rad }
      , // ... Remaining data members
   {}
```

OCP: A Procedural Approach

```
 private:
   ShapeType type;
};

class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : Shape{ circle }
      , radius{ rad }
      , // ... Remaining data members
   {}

   virtual ~Circle() = default;
   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

 private:
   double radius;
   // ... Remaining data members
};

void translate( Circle&, Vector3D const& );
void rotate( Circle&, Quaternion const& );
void draw( Circle const& );

class Square : public Shape
{
 public:
   explicit Square( double s )
      : Shape{ square }
      side{ s }
```

OCP: A Procedural Approach

```
void translate( Circle&, Vector3D const& );
void rotate( Circle&, Quaternion const& );
void draw( Circle const& );

class Square : public Shape
{
 public:
   explicit Square( double s )
      : Shape{ square }
      , side{ s }
      , // ... Remaining data members
   {}

   virtual ~Square() = default;
   double getSide() const noexcept;
   // ... getCenter(), getRotation(), ...

 private:
   double side;
   // ... Remaining data members
};

void translate( Square&, Vector3D const& );
void rotate( Square&, Quaternion const& );
void draw( Square const& );

void draw( std::vector<std::unique_ptr<Shape>> const& shapes )
{
   for( auto const& s : shapes )
   {
      switch ( s->getType() )
      {
```

OCP: A Procedural Approach

```
   private:
      double side;
      // ... Remaining data members
};

void translate( Square&, Vector3D const& );
void rotate( Square&, Quaternion const& );
void draw( Square const& );

void draw( std::vector<std::unique_ptr<Shape>> const& shapes )
{
   for( auto const& s : shapes )
   {
      switch ( s->getType() )
      {
         case circle:
            draw( *static_cast<Circle const*>( s.get() ) );
            break;
         case square:
            draw( *static_cast<Square const*>( s.get() ) );
            break;
      }
   }
}

int main()
{
   using Shapes = std::vector<std::unique_ptr<Shape>>;

   // Creating some shapes
   Shapes shapes;
   shapes.push_back( std::make_unique<Circle>( 2.0 ) );
   shapes.push_back( std::make_unique<Square>( 1.5 ) );
```

OCP: A Procedural Approach

```
            draw( *static_cast<Circle const*>( s.get() ) );
            break;
         case square:
            draw( *static_cast<Square const*>( s.get() ) );
            break;
      }
   }
}

int main()
{
   using Shapes = std::vector<std::unique_ptr<Shape>>;

   // Creating some shapes
   Shapes shapes;
   shapes.push_back( std::make_unique<Circle>( 2.0 ) );
   shapes.push_back( std::make_unique<Square>( 1.5 ) );
   shapes.push_back( std::make_unique<Circle>( 4.2 ) );

   // Drawing all shapes
   draw( shapes );
}
```

OCP: A Procedural Approach

```
enum ShapeType
{
   circle,
   square,
   rectangle
};

class Shape
{
 public:
   explicit Shape( ShapeType t )
      : type{ t }
   {}

   virtual ~Shape() = default;
   ShapeType getType() const noexcept;

 private:
   ShapeType type;
};

class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : Shape{ circle }
      , radius{ rad }
      , // ... Remaining data members
   {}
```

OCP: A Procedural Approach

```
 private:
   ShapeType type;
};

class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : Shape{ circle }
      , radius{ rad }
      , // ... Remaining data members
   {}

   virtual ~Circle() = default;
   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

 private:
   double radius;
   // ... Remaining data members
};

void translate( Circle&, Vector3D const& );
void rotate( Circle&, Quaternion const& );
void draw( Circle const& );

class Square : public Shape
{
 public:
   explicit Square( double s )
      : Shape{ square }
      side{ s }
```

OCP: A Procedural Approach

```
void translate( Circle&, Vector3D const& );
void rotate( Circle&, Quaternion const& );
void draw( Circle const& );

class Square : public Shape
{
 public:
   explicit Square( double s )
      : Shape{ square }
      , side{ s }
      , // ... Remaining data members
   {}

   virtual ~Square() = default;
   double getSide() const noexcept;
   // ... getCenter(), getRotation(), ...

 private:
   double side;
   // ... Remaining data members
};

void translate( Square&, Vector3D const& );
void rotate( Square&, Quaternion const& );
void draw( Square const& );

void draw( std::vector<std::unique_ptr<Shape>> const& shapes )
{
   for( auto const& s : shapes )
   {
      switch ( s->getType() )
      {
```

OCP: A Procedural Approach

```
void translate( Square&, Vector3D const& );
void rotate( Square&, Quaternion const& );
void draw( Square const& );

void draw( std::vector<std::unique_ptr<Shape>> const& shapes )
{
   for( auto const& s : shapes )
   {
      switch ( s->getType() )
      {
         case circle:
            draw( *static_cast<Circle const*>( s.get() ) );
            break;
         case square:
            draw( *static_cast<Square const*>( s.get() ) );
            break;
         case rectangle:
            draw( *static_cast<Rectangle const*>( s.get() ) );
            break;
      }
   }
}


int main()
{
   using Shapes = std::vector<std::unique_ptr<Shape>>;

   // Creating some shapes
   Shapes shapes;
   shapes.push_back( std::make_unique<Circle>( 2.0 ) );
   shapes.push_back( std::make_unique<Square>( 1.5 ) );
```

The Open-Closed Principle (OCP)

”This kind of type-based programming has a long history in C, and one of the things we know about it is that it yields programs that are essentially unmaintainable.”

(Scott Meyers, More Effective C++)

OCP: An Object-Oriented Approach

```
class Shape
{
 public:
   Shape() = default;
   virtual ~Shape() = default;

   virtual void translate( Vector3D const& ) = 0;
   virtual void rotate( Quaternion const& ) = 0;
   virtual void draw() const = 0;
};

class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

   virtual ~Circle() = default;

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

   void translate( Vector3D const& ) override;
   void rotate( Quaternion const& ) override;
   void draw() const override;

 private:
   double radius;
```

OCP: An Object-Oriented Approach

```
class Shape
{
 public:
   Shape() = default;
   virtual ~Shape() = default;

   virtual void translate( Vector3D const& ) = 0;
   virtual void rotate( Quaternion const& ) = 0;
   virtual void draw() const = 0;
};

class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

   virtual ~Circle() = default;

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

   void translate( Vector3D const& ) override;
   void rotate( Quaternion const& ) override;
   void draw() const override;

 private:
   double radius;
```

OCP: An Object-Oriented Approach

```
   virtual void translate( Vector3D const& ) = 0;
   virtual void rotate( Quaternion const& ) = 0;
   virtual void draw() const = 0;
};

class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

   virtual ~Circle() = default;

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

   void translate( Vector3D const& ) override;
   void rotate( Quaternion const& ) override;
   void draw() const override;

 private:
   double radius;
   // ... Remaining data members
};

class Square : public Shape
{
 public:
   explicit Square( double s )
      : side{ s }
```

OCP: An Object-Oriented Approach

```
 private:
   double radius;
   // ... Remaining data members
};

class Square : public Shape
{
 public:
   explicit Square( double s )
      : side{ s }
      , // ... Remaining data members
   {}

   virtual ~Square() = default;

   double getSide() const noexcept;
   // ... getCenter(), getRotation(), ...

   void translate( Vector3D const& ) override;
   void rotate( Quaternion const& ) override;
   void draw() const override;

 private:
   double side;
   // ... Remaining data members
};

void draw( std::vector<std::unique_ptr<Shape>> const& shapes )
{
   for( auto const& s : shapes )
   {
      s->draw()
```

OCP: An Object-Oriented Approach

```
   // ... getCenter(), getRotation(), ...

   void translate( Vector3D const& ) override;
   void rotate( Quaternion const& ) override;
   void draw() const override;

 private:
   double side;
   // ... Remaining data members
};

void draw( std::vector<std::unique_ptr<Shape>> const& shapes )
{
   for( auto const& s : shapes )
   {
      s->draw()
   }
}

int main()
{
   using Shapes = std::vector<std::unique_ptr<Shape>>;

   // Creating some shapes
   Shapes shapes;
   shapes.push_back( std::make_unique<Circle>( 2.0 ) );
   shapes.push_back( std::make_unique<Square>( 1.5 ) );
   shapes.push_back( std::make_unique<Circle>( 4.2 ) );

   // Drawing all shapes
   draw( shapes );
}
```

OCP: An Object-Oriented Approach

```
void draw( std::vector<std::unique_ptr<Shape>> const& shapes )
{
   for( auto const& s : shapes )
   {
      s->draw()
   }
}

int main()
{
   using Shapes = std::vector<std::unique_ptr<Shape>>;

   // Creating some shapes
   Shapes shapes;
   shapes.push_back( std::make_unique<Circle>( 2.0 ) );
   shapes.push_back( std::make_unique<Square>( 1.5 ) );
   shapes.push_back( std::make_unique<Circle>( 4.2 ) );

   // Drawing all shapes
   draw( shapes );
}
```

OCP: An Object-Oriented Approach

```
   virtual void translate( Vector3D const& ) = 0;
   virtual void rotate( Quaternion const& ) = 0;
   virtual void draw() const = 0;
};

class Circle : public Shape
{
 public:
   explicit Circle( double rad )
      : radius{ rad }
      , // ... Remaining data members
   {}

   virtual ~Circle() = default;

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

   void translate( Vector3D const& ) override;
   void rotate( Quaternion const& ) override;
   void draw() const override;

 private:
   double radius;
   // ... Remaining data members
};

class Square : public Shape
{
 public:
   explicit Square( double s )
      : side{ s }
```


CPP
EUROPE
Klaus Iglberger
Embrace No-Paradigm Programming!
2020 FEB 25, Bucharest | www.cppeurope.com
0:00 / 1:03:30

Design Evaluation

	Addition of shapes (OCP)	Addition of operations (OCP)	Separation of Concerns (SRP)	Ease of Use	Performance
Enum	1	7	5	6	9
OO	8	2	2	6	6
Visitor	2	8	8	3	1
mpark::variant	3	9	9	9	9
Strategy	7	2	7	4	2
std::function	8	3	7	7	5
Type Erasure	9	4	8	8	6

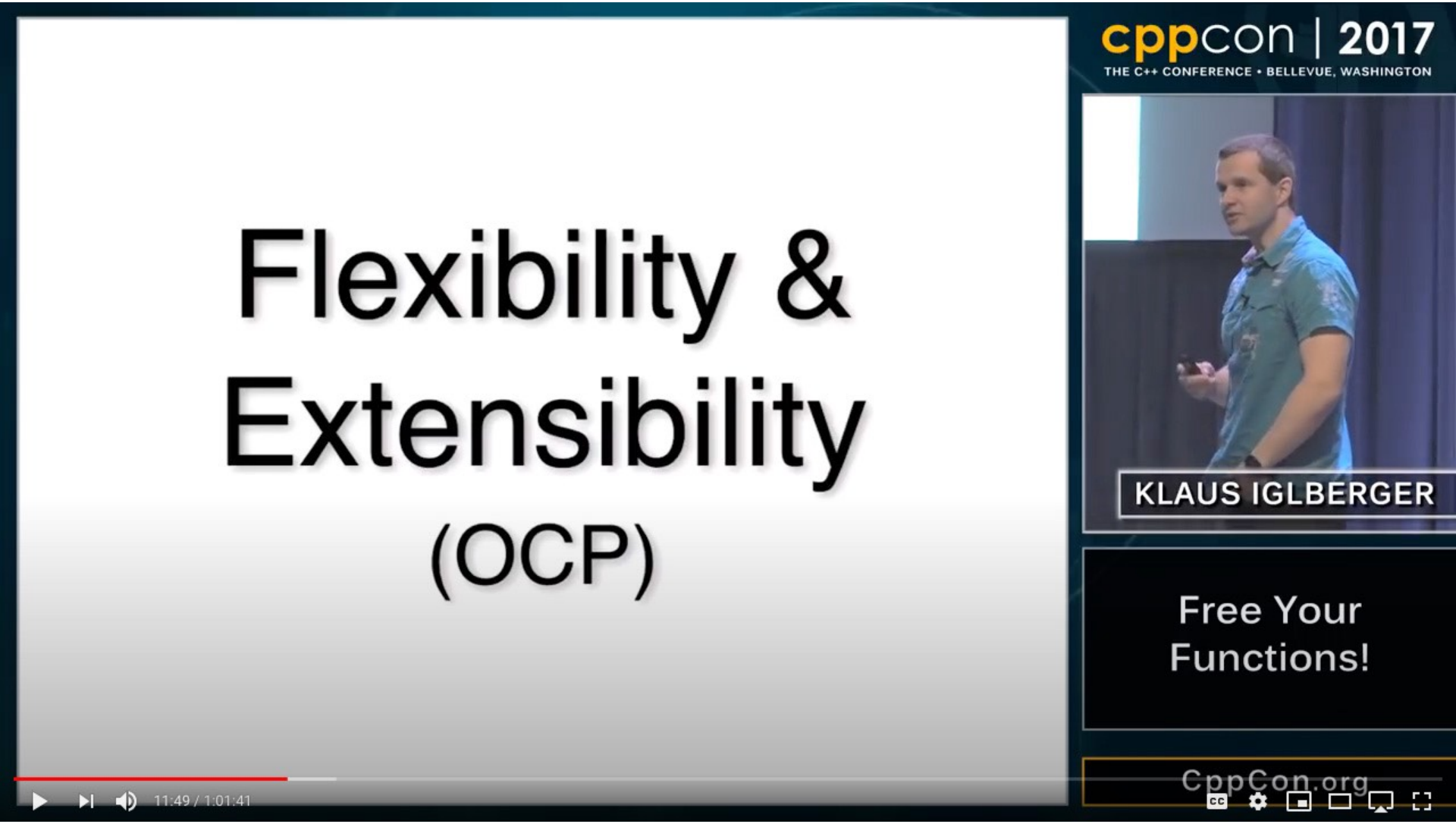
1 = very bad, …, 9 = very good

The Open-Closed Principle (OCP)

```
namespace std {

template< typename InputIt, typename OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt dest )
{
   for( ; first != last; ++first, ++dest ) {
      *dest = *first;
   }

   return dest;
}

} // namespace std
```

The **`copy()`** function works for all copyable types. It …

- … works for all types that adhere to the required concepts;
- … does not have to be modified for new types.


Flexibility &
Extensibility
(OCP)
cppcon | 2017
THE C++ CONFERENCE • BELLEVUE, WASHINGTON
KLAUS IGLBERGER
Free Your
Functions!
CppCon.org
11:49 / 1:01:41

The Open-Closed Principle (OCP)

Guideline: Prefer software design that allows the addition of types or operations without the need to modify existing code.

The Liskov Substitution Principle (LSP)

The Liskov Substitution Principle (LSP)

”What is wanted here is something like the following substitution property: If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T.”

(Barbara Liskov, Data Abstraction and Hierarchy)

Or in simplified form:

”Subtypes must be substitutable for their base types.”

The Liskov Substitution Principle (LSP)

Behavioral subtyping (aka "IS-A" relationship)

- Contravariance of **method arguments** in a subtype
- Covariance of **return types** in a subtype
- **Preconditions** cannot be strengthened in a subtype
- **Postconditions** cannot be weakened in a subtype
- **Invariants** of the super type must be preserved in a subtype

The Liskov Substitution Principle (LSP)

Which of the following two implementations would you choose?

```
//***** Option A *****

class Square
{
 public:
   virtual void setWidth(double);
   virtual int getArea();
   // ...
 private:
   double width;
};

class Rectangle
   : public Square
{
 public:
   virtual void setHeight(double);
   // ...
 private:
   double height;
};
```

```
//***** Option B *****

class Rectangle
{
 public:
   virtual void setWidth(double);
   virtual void setHeight(double);
   virtual int getArea();
   // ...
 private:
   double width;
   double height;
};

class Square
   : public Rectangle
{
   // ...
};
```

The Liskov Substitution Principle (LSP)

```
namespace std {

template< typename InputIt, typename OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt dest )
{
   for( ; first != last; ++first, ++dest ) {
      *dest = *first;
   }

   return dest;
}

} // namespace std
```

The Liskov Substitution Principle (LSP)

```
namespace std {

template< typename InputIt, typename OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt dest )
{
   for( ; first != last; ++first, ++dest ) {
      *dest = *first;
   }

   return dest;
}

} // namespace std
```

The **copy()** function works if …

- … the given InputIt adheres to the required concept;
- … the given OutputIt adheres to the required concept.

The Liskov Substitution Principle (LSP)

Guideline: Make sure that inheritance is about behavior, not about data.

Guideline: Make sure that the contract of base types is adhered to.

Guideline: Make sure to adhere to the required concept.

The Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP)

”Clients should not be forced to depend on methods that they do not use.”

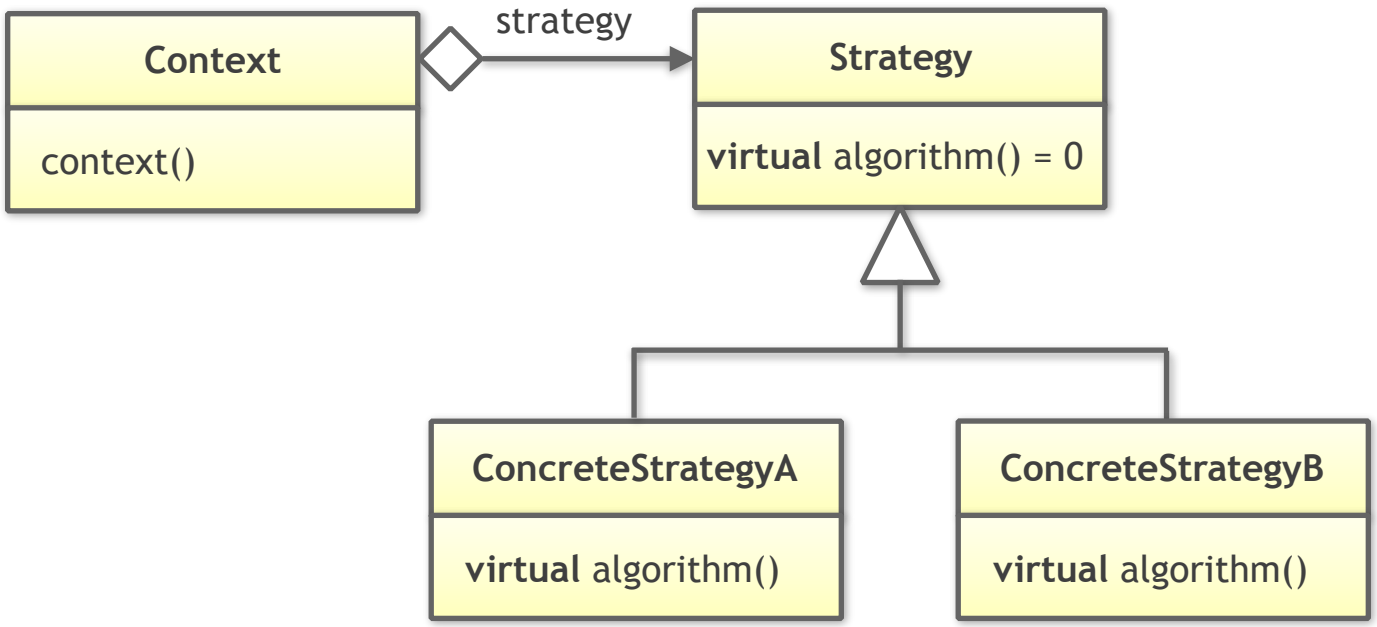
(Robert C. Martin, Agile Software Development)

The Interface Segregation Principle (ISP)

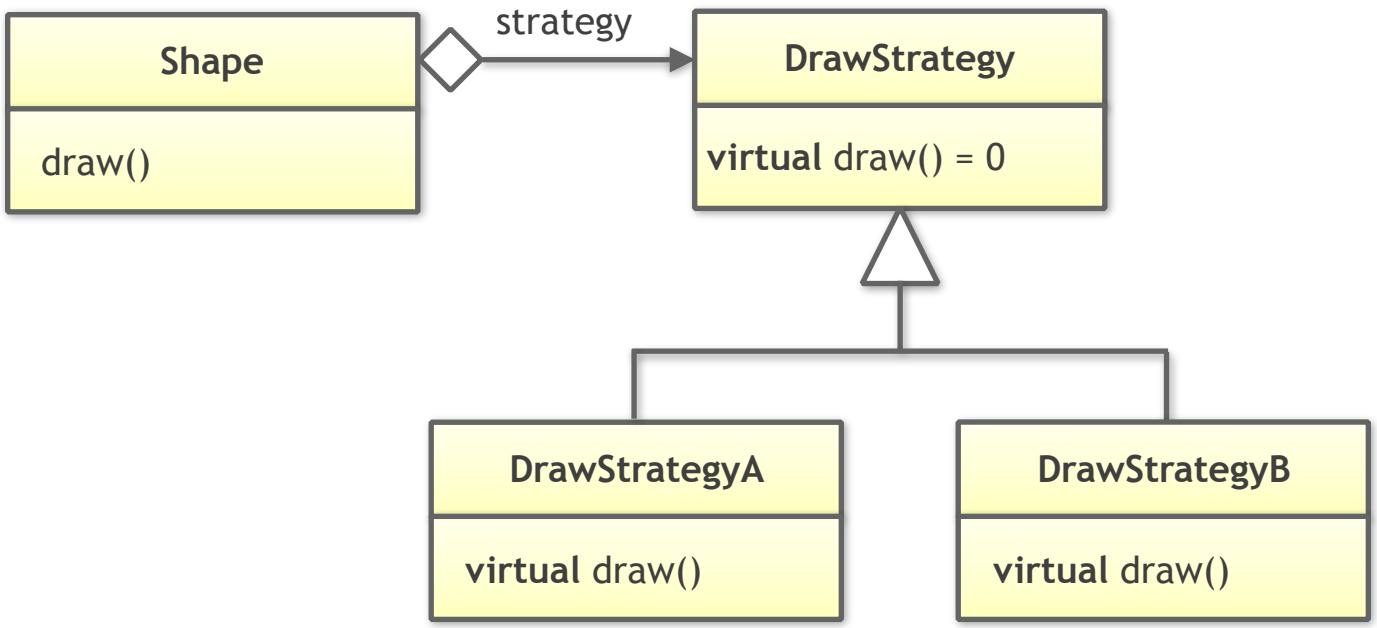
”Many client specific interfaces are better than one general-purpose interface.”

(Wikipedia)

The Interface Segregation Principle (ISP)



The Interface Segregation Principle (ISP)



The Interface Segregation Principle (ISP)

```
class Circle;
class Square;

class DrawStrategy
{
 public:
   virtual ~DrawStrategy() {}

   virtual void draw( const Circle& circle ) const = 0;
   virtual void draw( const Square& square ) const = 0;
};


class Shape
{
 public:
   Shape() = default;
   virtual ~Shape() = default;

   virtual void translate( Vector3D const& ) = 0;
   virtual void rotate( Quaternion const& ) = 0;
   virtual void draw() const = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad, std::unique_ptr<DrawStrategy> ds )
      : radius{ rad }
```

The Interface Segregation Principle (ISP)

```
class Circle;
class Square;

class DrawStrategy
{
 public:
   virtual ~DrawStrategy() {}

   virtual void draw( const Circle& circle ) const = 0;
   virtual void draw( const Square& square ) const = 0;
};


class Shape
{
 public:
   Shape() = default;
   virtual ~Shape() = default;

   virtual void translate( Vector3D const& ) = 0;
   virtual void rotate( Quaternion const& ) = 0;
   virtual void draw() const = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad, std::unique_ptr<DrawStrategy> ds )
      : radius{ rad }
```

The Interface Segregation Principle (ISP)

```
class DrawStrategy
{
 public:
   virtual ~DrawStrategy() {}

   virtual void draw( const Circle& circle ) const = 0;
   virtual void draw( const Square& square ) const = 0;
};


class Shape
{
 public:
   Shape() = default;
   virtual ~Shape() = default;

   virtual void translate( Vector3D const& ) = 0;
   virtual void rotate( Quaternion const& ) = 0;
   virtual void draw() const = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad, std::unique_ptr<DrawStrategy> ds )
      : radius{ rad }
      , // ... Remaining data members
      , drawing{ std::move(ds) }
   {}

   virtual ~Circle() = default;
```

The Interface Segregation Principle (ISP)

```
   virtual void draw() const = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad, std::unique_ptr<DrawStrategy> ds )
      : radius{ rad }
      , // ... Remaining data members
      , drawing{ std::move(ds) }
   {}

   virtual ~Circle() = default;

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

   void translate( Vector3D const& ) override;
   void rotate( Quaternion const& ) override;
   void draw() const override;

 private:
   double radius;
   // ... Remaining data members
   std:unique_ptr<DrawStrategy> drawing;
};



class Square : public Shape
{
 public:
```

The Interface Segregation Principle (ISP)

```
class Circle;
class Square;

class DrawStrategy
{
 public:
   virtual ~DrawStrategy() {}

   virtual void draw( const Circle& circle ) const = 0;
   virtual void draw( const Square& square ) const = 0;
};


class Shape
{
 public:
   Shape() = default;
   virtual ~Shape() = default;

   virtual void translate( Vector3D const& ) = 0;
   virtual void rotate( Quaternion const& ) = 0;
   virtual void draw() const = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad, std::unique_ptr<DrawStrategy> ds )
      : radius{ rad }
```

The Interface Segregation Principle (ISP)

```
class Circle;
class Square;

class DrawCircleStrategy
{
 public:
   virtual ~DrawCircleStrategy() {}

   virtual void draw( const Circle& circle ) const = 0;
};

class DrawSquareStrategy
{
 public:
   virtual ~DrawSquareStrategy() {}

   virtual void draw( const Square& square ) const = 0;
};

class Shape
{
 public:
   Shape() = default;
   virtual ~Shape() = default;

   virtual void translate( Vector3D const& ) = 0;
   virtual void rotate( Quaternion const& ) = 0;
   virtual void draw() const = 0;
};
```

The Interface Segregation Principle (ISP)

```
   virtual void rotate( Quaternion const& ) = 0;
   virtual void draw() const = 0;
};


class Circle : public Shape
{
 public:
   explicit Circle( double rad, std::unique_ptr<DrawCircleStrategy> ds )
      : radius{ rad }
      , // ... Remaining data members
      , drawing{ std::move(ds) }
   {}

   virtual ~Circle() = default;

   double getRadius() const noexcept;
   // ... getCenter(), getRotation(), ...

   void translate( Vector3D const& ) override;
   void rotate( Quaternion const& ) override;
   void draw() const override;

 private:
   double radius;
   // ... Remaining data members
   std:unique_ptr<DrawCircleStrategy> drawing;
};



class Square : public Shape
{
```

The Interface Segregation Principle (ISP)

```
namespace std {

template< typename InputIt, typename OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt dest )
{
   for( ; first != last; ++first, ++dest ) {
      *dest = *first;
   }

   return dest;
}

} // namespace std
```

The Interface Segregation Principle (ISP)

```
namespace std {

template< typename InputIt, typename OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt dest )
{
   for( ; first != last; ++first, ++dest ) {
      *dest = *first;
   }

   return dest;
}

} // namespace std
```

The **copy()** function …

- … only requires InputIt (minimum requirements) and …
- … only requires OutputIt (minimum requirements)

… and by that imposes minimum dependencies.

The Interface Segregation Principle (ISP)

Guideline: Make sure interfaces don’t induce unnecessary dependencies.

The Dependency Inversion Principle (DIP)

The Dependency Inversion Principle (DIP)

”The Dependency Inversion Principle (DIP) tells us that the most flexible systems are those in which source code dependencies refer only to abstractions, not to concretions.”

(Robert C. Martin, Clean Architecture)

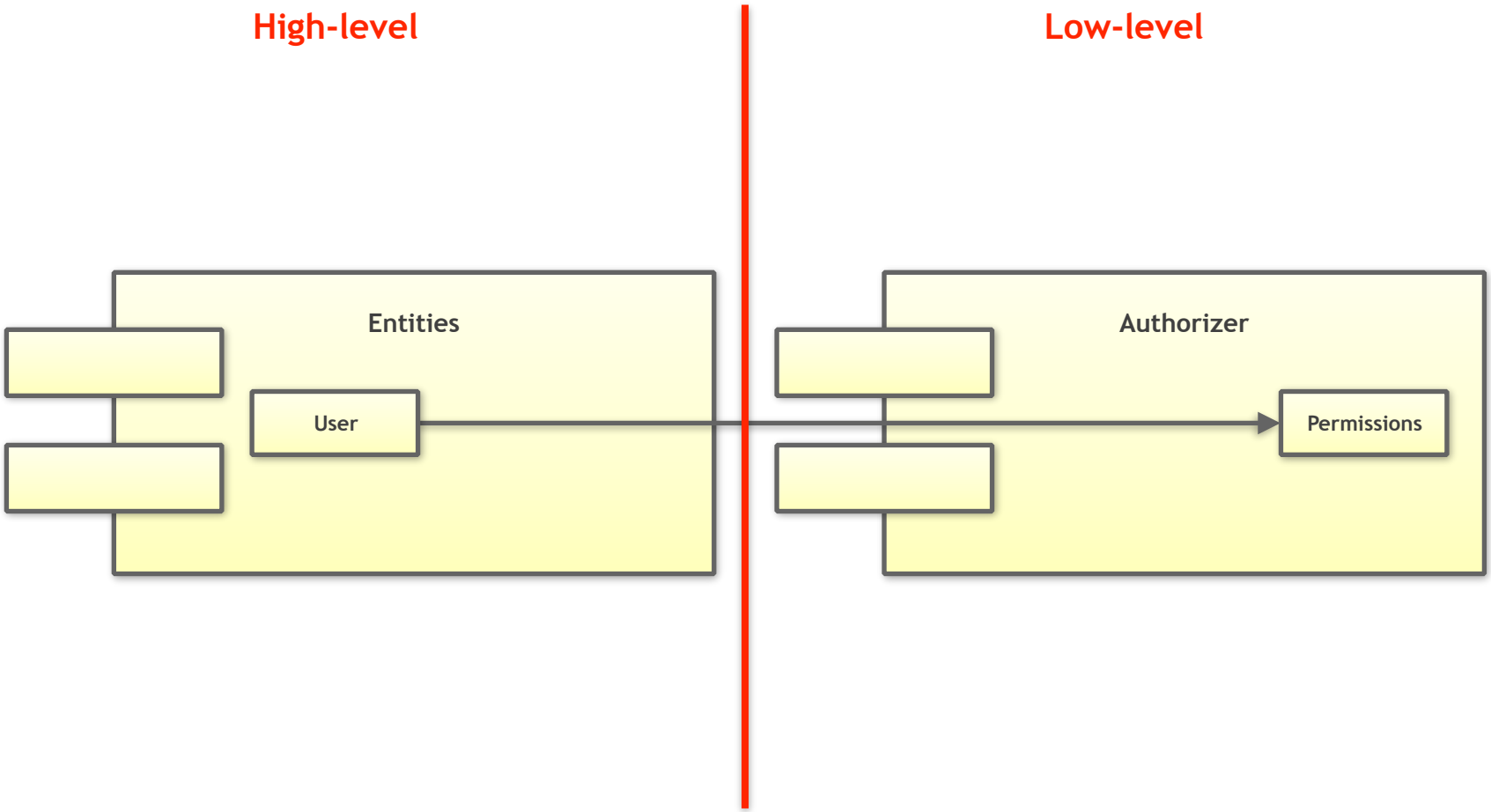
The Dependency Inversion Principle (DIP)

”a. High-level modules should not depend on low-level modules. Both should depend on abstractions.

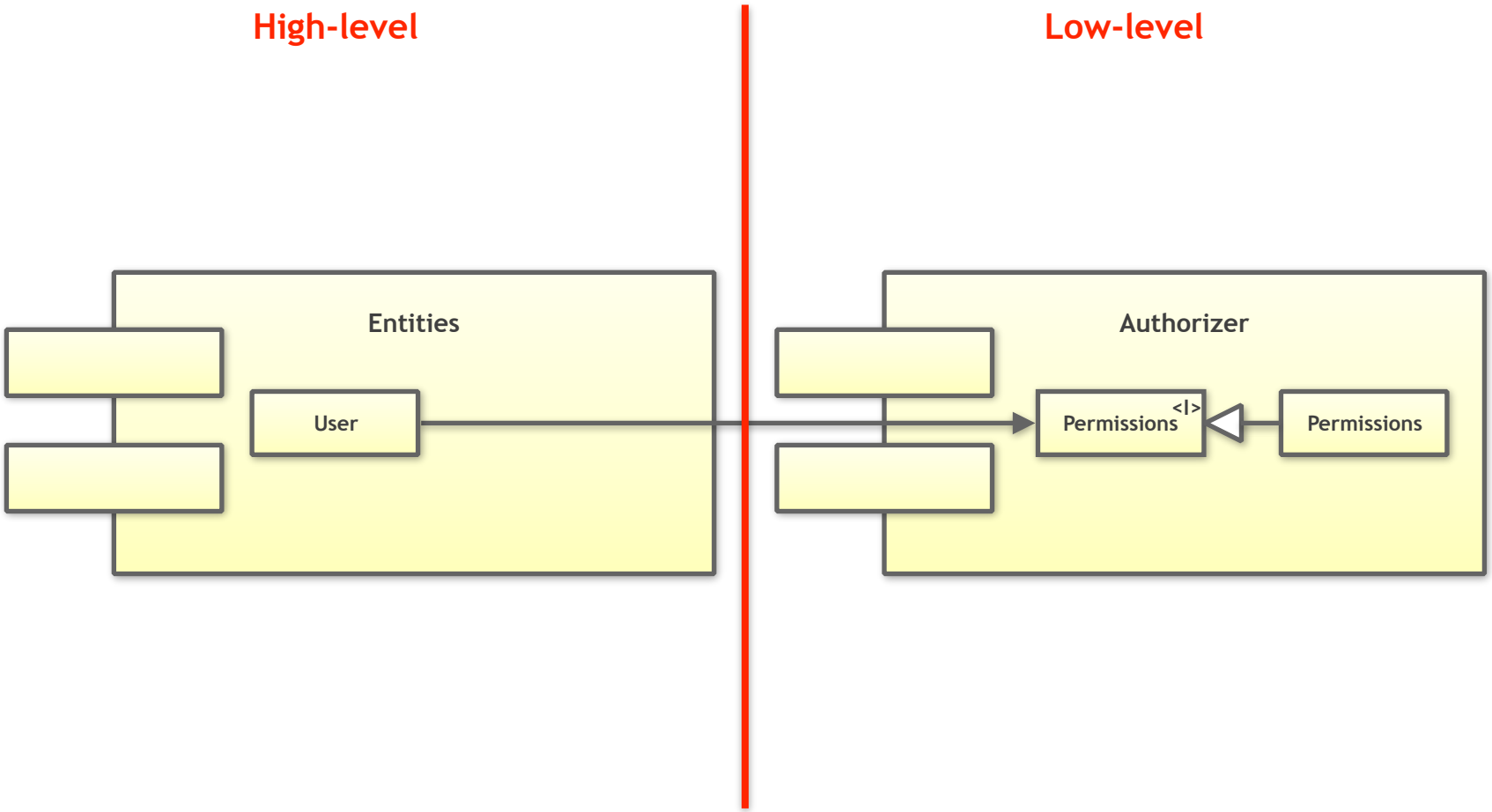
b. Abstractions should not depend on details. Details should depend on abstractions.”

(Robert C. Martin, Agile Software Development)

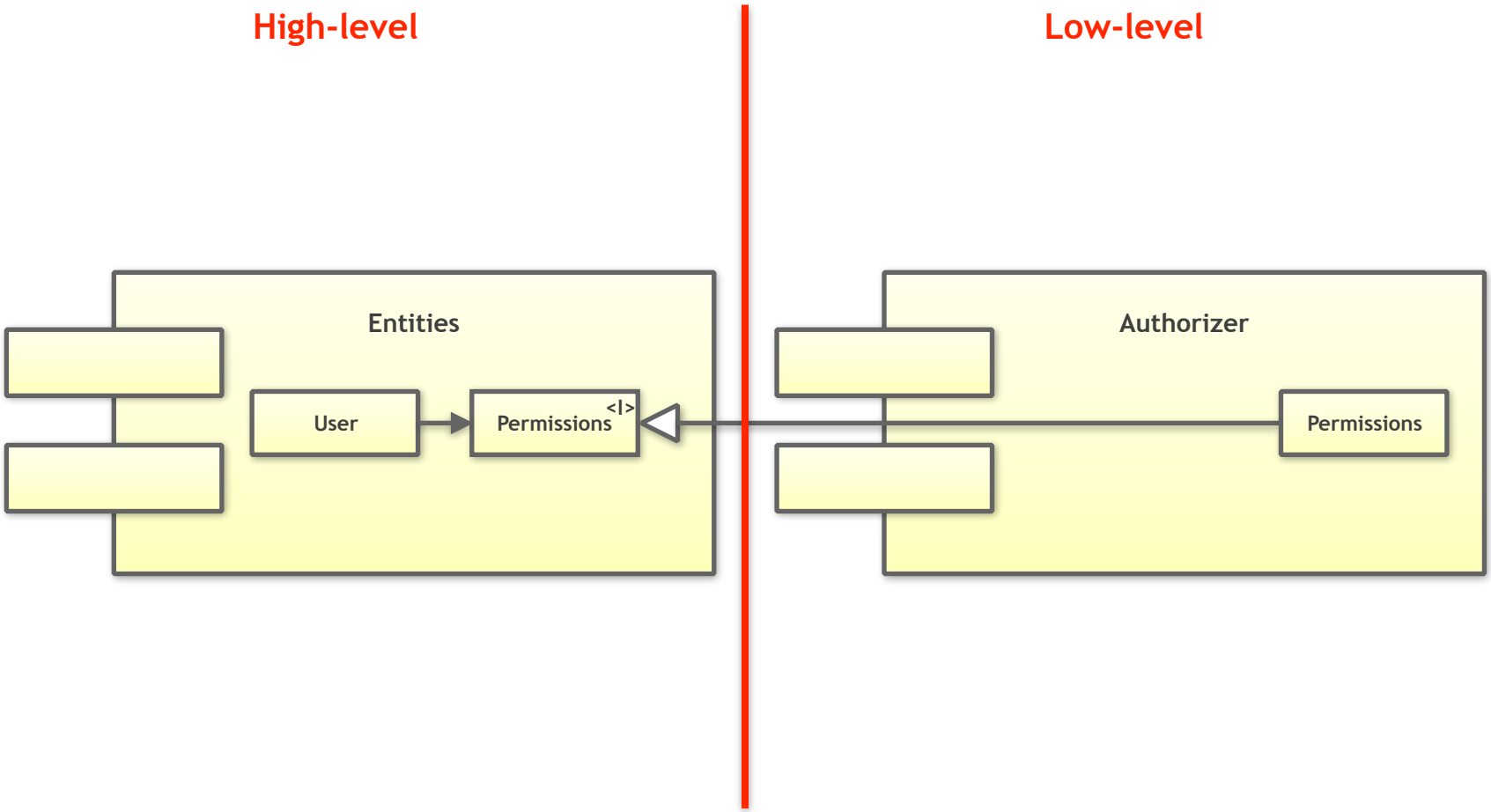
The Dependency Inversion Principle (DIP)



The Dependency Inversion Principle (DIP)



The Dependency Inversion Principle (DIP)



The Dependency Inversion Principle (DIP)

Model–view–controller

From Wikipedia, the free encyclopedia

Model–view–controller (usually known as MVC) is a software design pattern[1] commonly used for developing user interfaces which divides the related program logic into three interconnected elements. This is done to separate internal representations of information from the ways information is presented to and accepted from the user.[2][3] This kind of pattern is used for designing the layout of the page.

Traditionally used for desktop graphical user interfaces (GUIs), this pattern has become popular for designing web applications.[4] Popular programming languages like JavaScript, Python, Ruby, PHP, Java, C#, and Swift have MVC frameworks that are used for web or mobile application development straight out of the box.

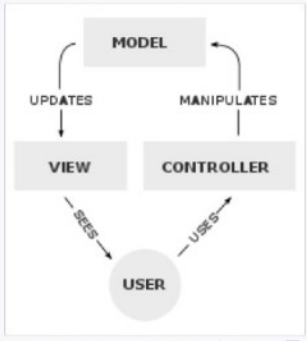
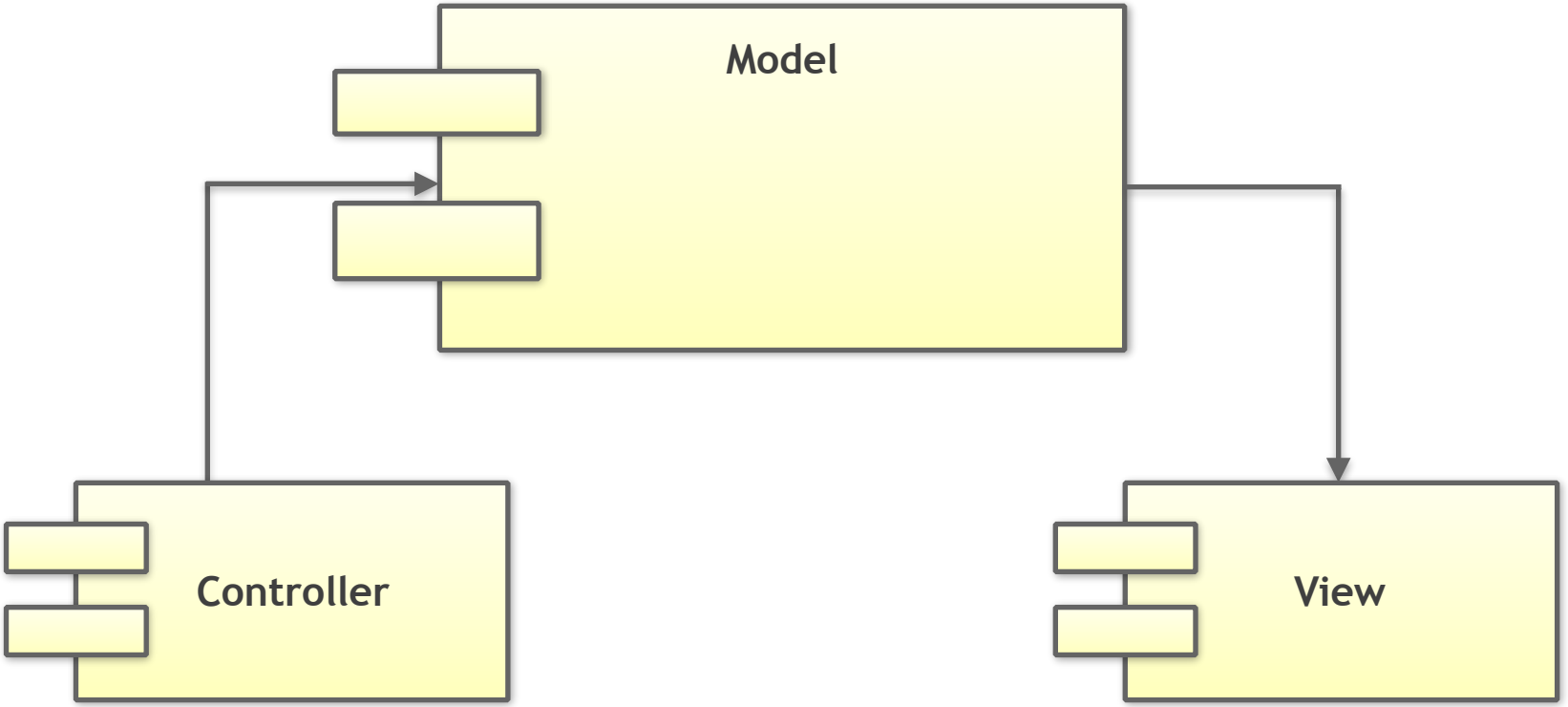


Diagram of interactions within the MVC pattern.

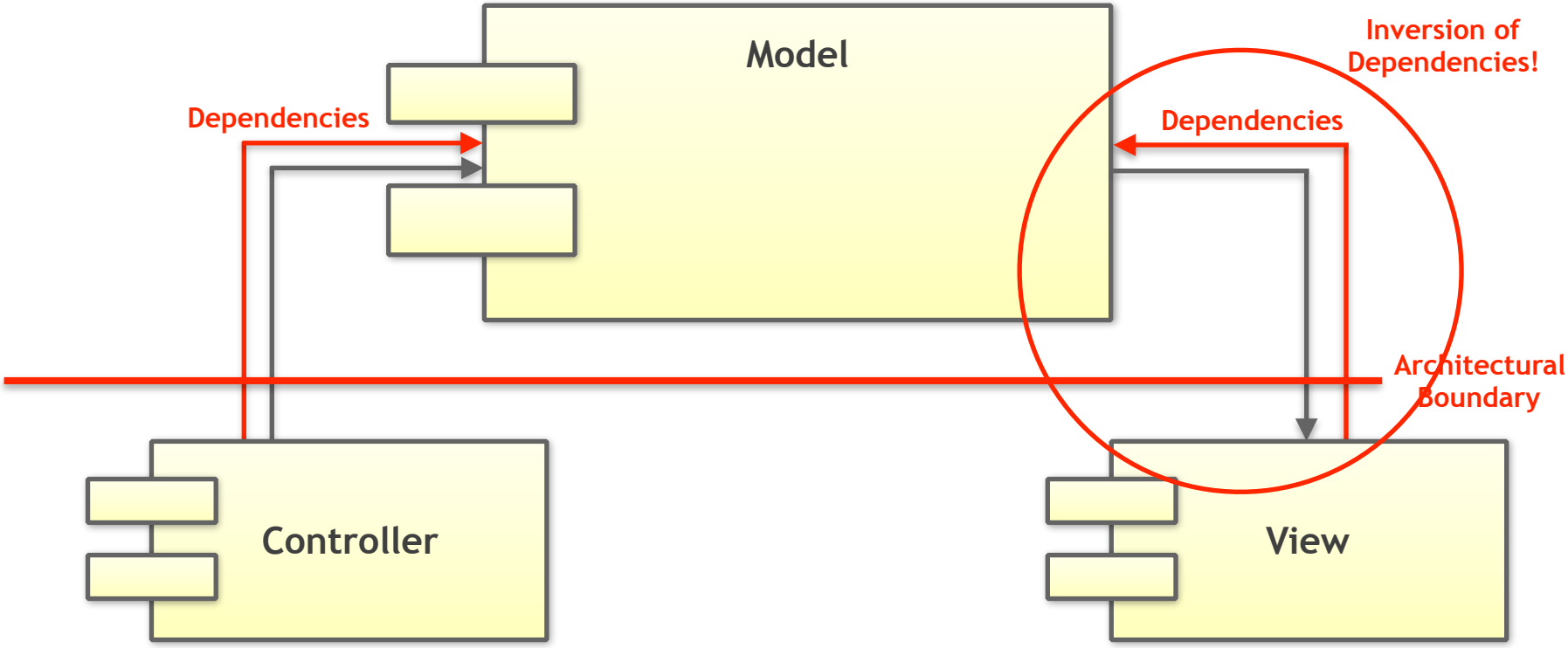
Components [edit]

Model

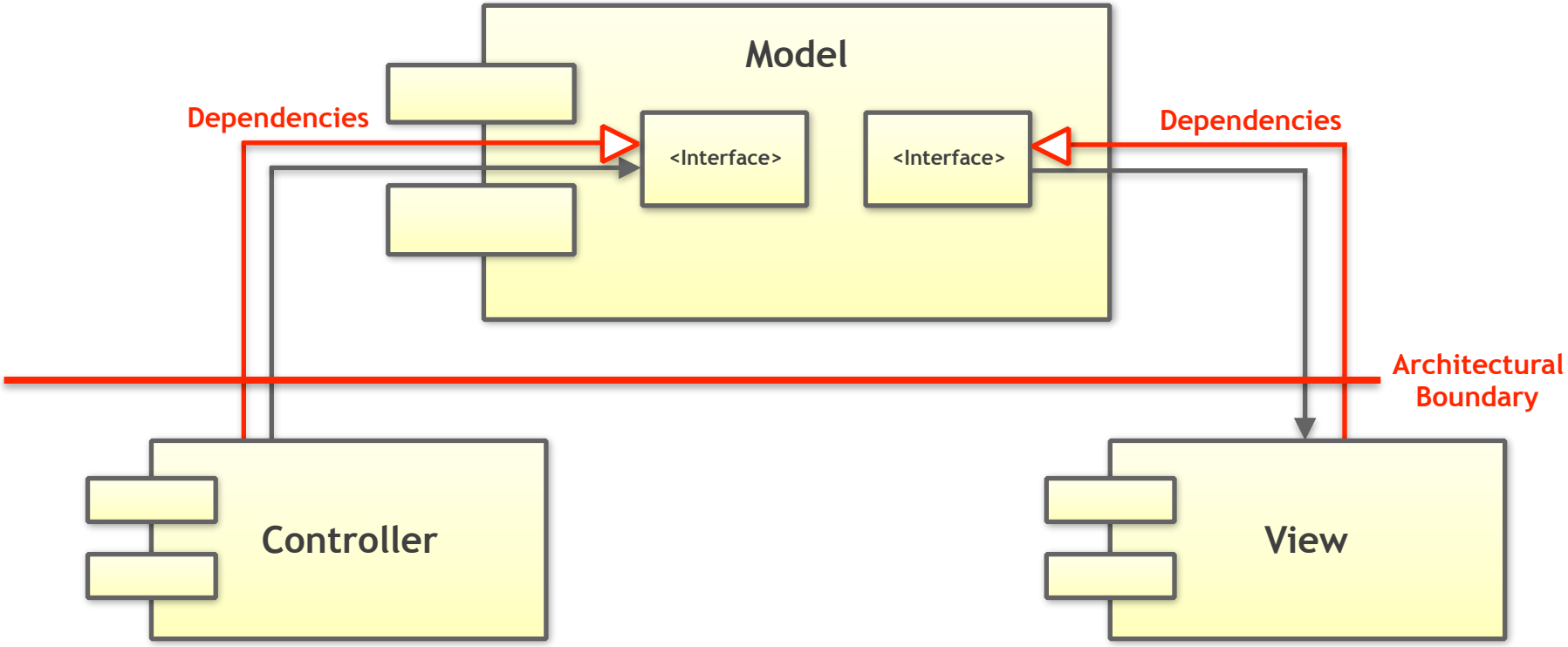
The Dependency Inversion Principle (DIP)



The Dependency Inversion Principle (DIP)



The Dependency Inversion Principle (DIP)



The Dependency Inversion Principle (DIP)

```
namespace std {

template< typename InputIt, typename OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt dest )
{
   for( ; first != last; ++first, ++dest ) {
      *dest = *first;
   }

   return dest;
}

} // namespace std
```

The Dependency Inversion Principle (DIP)

```
namespace std {

template< typename InputIt, typename OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt dest )
{
   for( ; first != last; ++first, ++dest ) {
      *dest = *first;
   }

   return dest;
}

} // namespace std
```

The **copy()** function …

- … is in control of its own requirements (concepts);
- … is implemented in terms of these requirements;
- … you depend on **copy()**, not **copy()** on you (dependency inversion).

The Dependency Inversion Principle (DIP)

Guideline: Prefer to depend on abstractions (i.e. abstract classes or concepts) instead of concrete types.

The SOLID Principles

Single-Responsibility Principle

Open-Closed Principle

Liskov Substitution Principle

Interface Segregation Principle

Dependency Inversion Principle

Summary

- The SOLID principles are more than just a set of OO guidelines
- Use the SOLID principles to reduce coupling and facilitate change
  - Separate concerns via the SRP to isolate changes
  - Design by OCP to simplify additions/extensions
  - Adhere to the LSP when using abstractions
  - Minimize the dependencies of interfaces via the ISP
  - Introduce abstractions to steer dependencies (DIP)

The SOLID Principles

Klaus Iglberger, CppEurope, Online Edition

klaus.iglberger@gmx.de