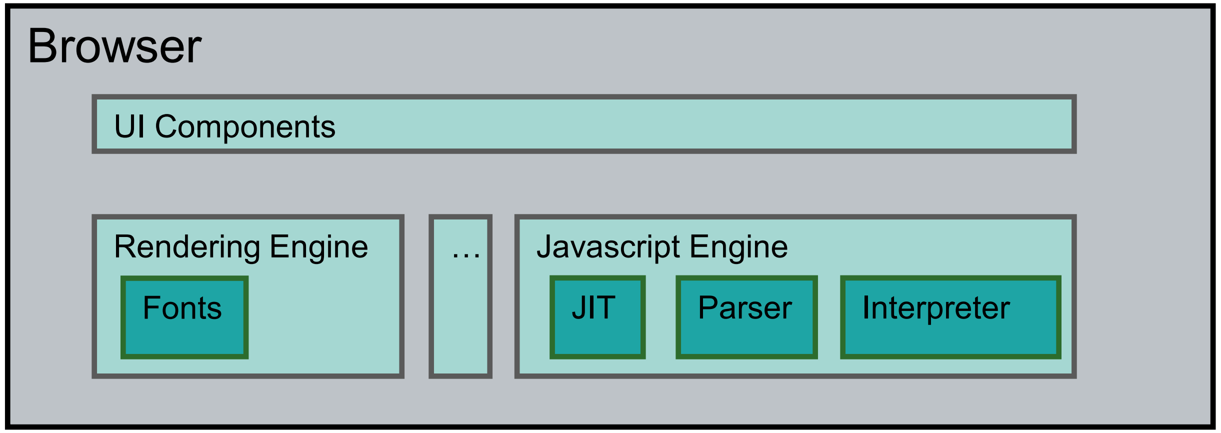
Software Engineering (D-MATH CSE)

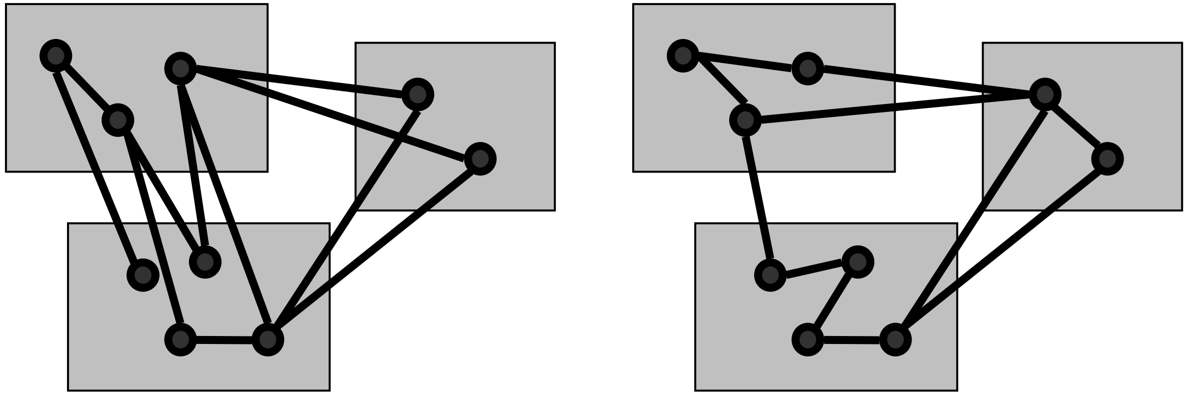
Modularity II

Marcel Lüthi, **Malte Schwerhoff**

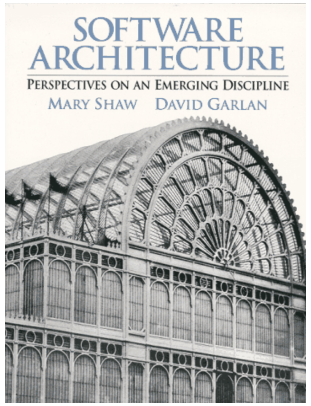
Slides based on Software Engineering by Peter Müller

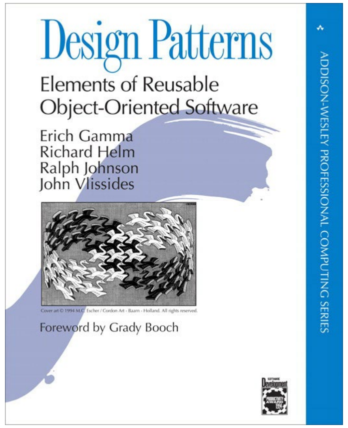
Last time





Architecture: Decomposing the system into modules
Design: Designing the modules



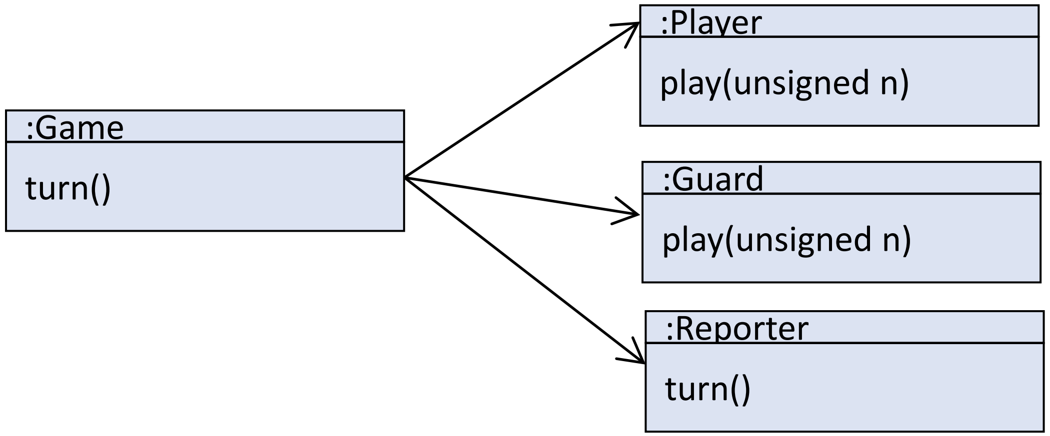


Coupling problems

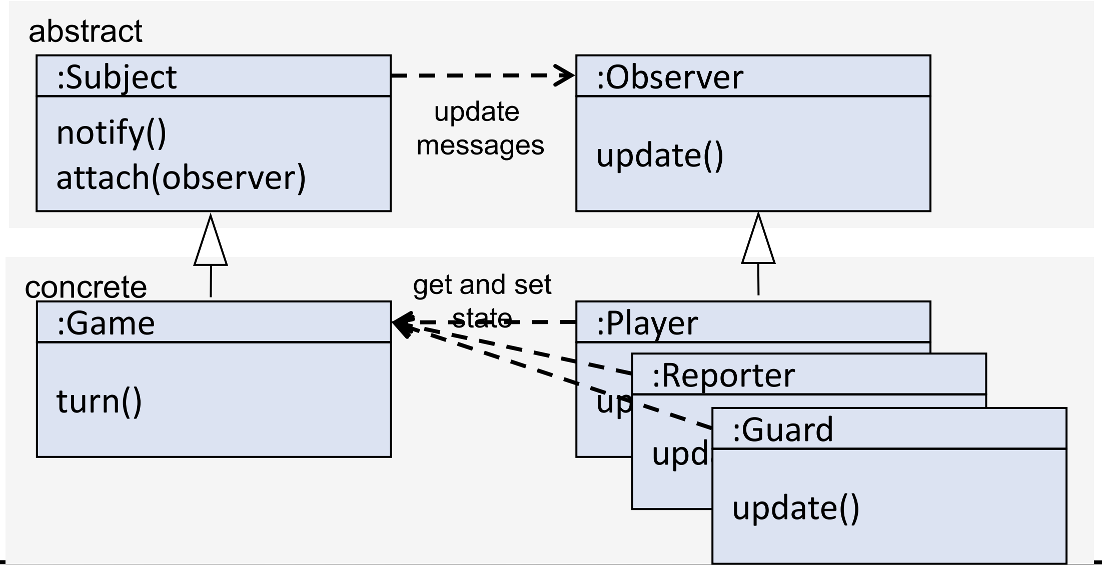
- Data Coupling
- Procedural Coupling

Reminder: Observer Pattern

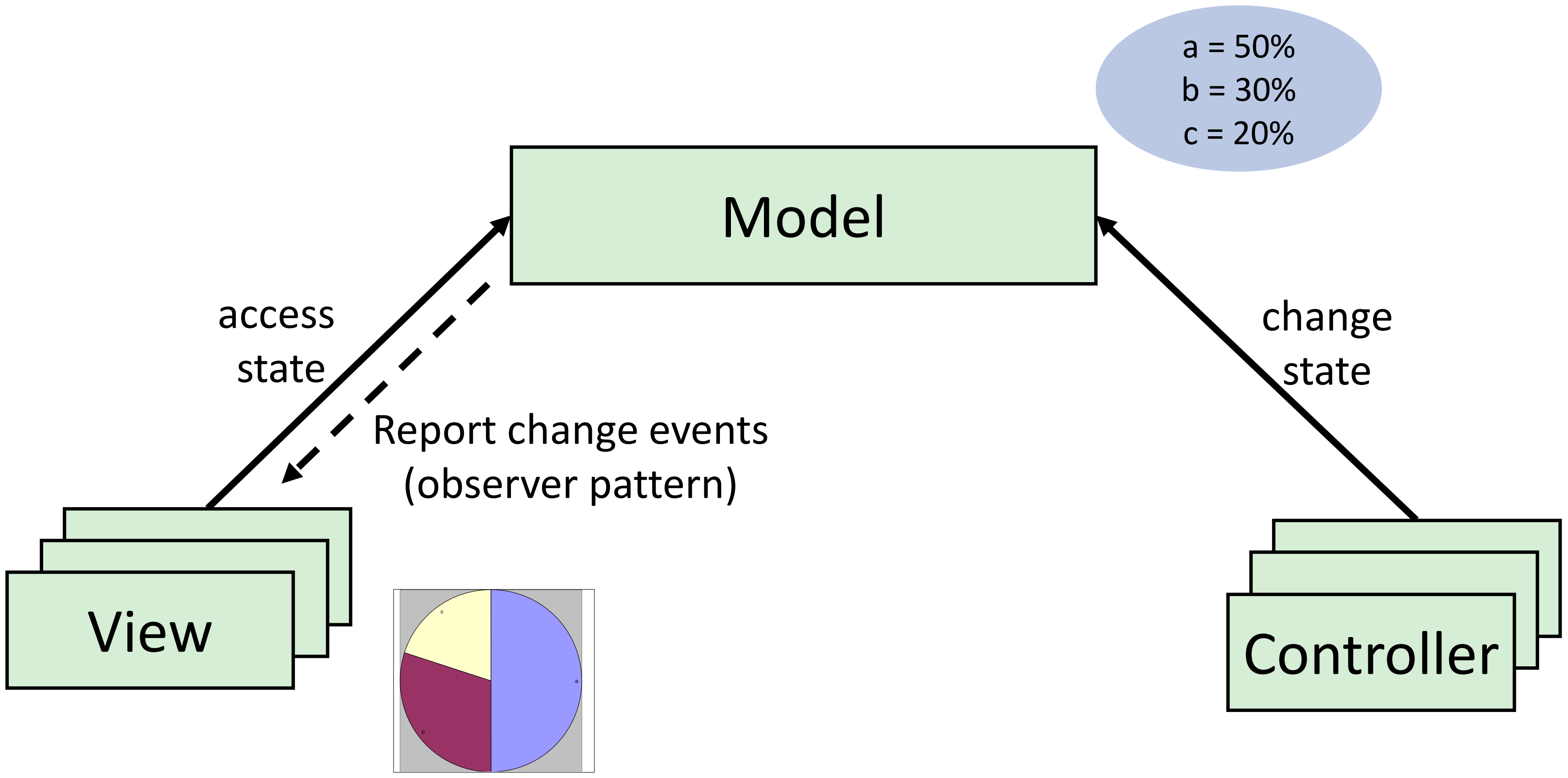
```
class Game {
  ...
   void turn(){
     reporter->turn();
     for (auto p: players)
       p->play(onTurn);
     guard->play(onTurn);
     onTurn = (onTurn+1) % players.size();
   }
};
```




```
class Game : Subject {

unsigned current_turn(){
    return onTurn;}
 void turn(){
    notify();
    onTurn = (onTurn+1) % players;
 }
};
```



Reminder: MVC & Observer pattern



Code example on code expert!

Discussion

The observer pattern reduces coupling by replacing static procedure calls by sending events

- Why does this add flexibility? What can we do that was not possible before?
- What is the disadvantage of this solution?

Today’s program

- **More Coupling problems**
  - **Class Coupling**
- Adaptation (how to create code that can react to change)

Class Coupling: Sources, Problems, Solutions 1

😐 Classes depend on other classes via types: member variables, function signatures, local variables

☹ Problem: Inhibits code reuse due to specific type

```
class Printer {
  vector<int> data;
public:
  Printer(const vector<int>& v): data(v) {}
  void print() const {
    for (int e : data) {
      cout << e << " ";
    }
  }
}
```

☺ Solution: Abstract over concrete types

- E.g. iterator(s) instead of vector
- Abstract superclasses, interfaces
- Templates (to be discussed later), generics

Class Coupling: Sources, Problems, Solutions 2

😐 Class dependencies via inheritance

```
class Sub : public Sup1, Sup2 {
  ...
  int compute() {
    // bar1() inherited from Sup1
    return foo() * bar1();
  }
}
```

☹ Problems:

1. Multiple inheritance
   intricate correctness details
2. Changes in superclass may break subclass (*fragile base class problem*)

☺ One solution (not always possible):

Avoid problem in the first place:
refactor inheritance to subtyping + aggregation + delegation

Code example – Fragile base class problem

```
class MySet {
  std::set<std::string> data;

public:
    virtual bool add(const std::string& element) {
        return data.insert(element).second;
    }

    virtual void addAll(const std::vector<std::string>& vec) {
        for (const auto& element : vec) {
            //this->add(element);
            data.insert(element);
        }
    }
};
```

`this->add(elem)` and `data.insert(elem)` seem interchangeable

Also see this example on Code Expert under "Code examples"

Code example – Fragile base class problem

```
class CountingSet : public MySet {
   int addCount = 0;

public:
    bool add(const std::string& element) override {
        addCount++;
        return MySet::add(element);
    }

    void addAll(const std::vector<std::string>& vec) override {
        addCount += vec.size();
        MySet::addAll(vec);
    }

    int getAddCount() const {
        return addCount;
    }
};
```

Extends `MySet`

Add counting functionality

Add counting functionality

What happens when you change, *in the superclass*, `addAll` to use `add` instead of `insert`?

Fragile base class problem: Solution

```cpp
class CountingSet {
    MySet mySet;
    int addCount = 0;

public:
    bool add(const std::string& element) override {
        addCount++;
        return mySet.add(element);
    }

    void addAll(const std::vector<std::string>& vec) override {
        addCount += vec.size();
        mySet.addAll(vec);
    }

    int getAddCount() const {
        return addCount;
    }
};
```

Aggregate MySet (instead of extending it)

Add counting functionality

Add counting functionality

Rule of thumb: favour aggregation over inheritance

Class Coupling: Sources, Problems, Solutions 3

😐 Class dependencies via object creation

☹ Problems:

- Reuse
- Difficult to test in isolation (*stubs*, *mocking*)

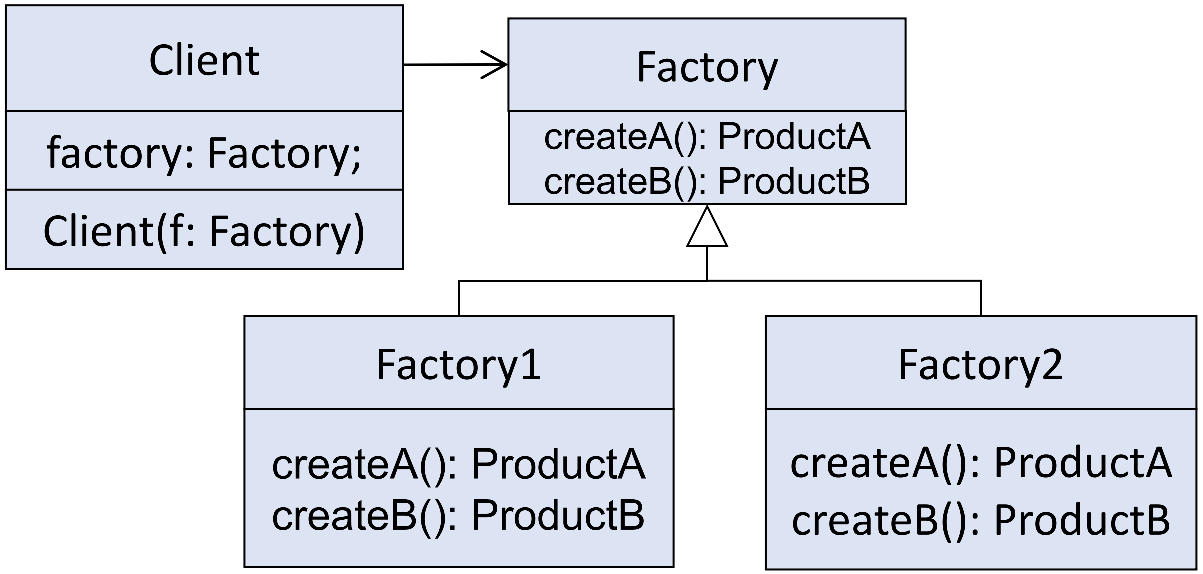
☺ Solution: externalize object creation

- Constructor parameters
- *Factory method* pattern, *abstract factory* pattern
- *Dependency injection*

```
class App {
  auto database = MongoDB(...);
        ...
}
```

```
class App {
  Database& database;
public:
  App(Database& db) {
    database = db;
  }
}
```

Factory Method Pattern



- The *factory method* pattern is a classical object-oriented pattern for abstracting over object creation
- Together with inheritance it allows to abstract away completely from the particularities and dependencies of a class

Factory Method Example

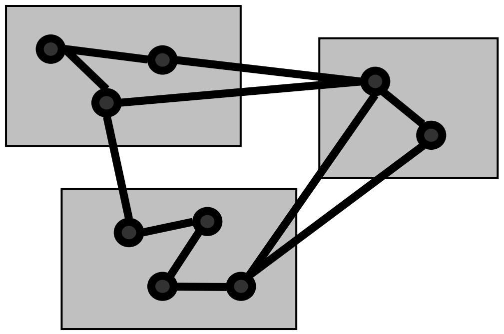
```
class DatabaseFactory {
public:
  virtual Connection* createConnection() const = 0;
 ...
};
```

```
class MongoDBFactory : public DatabaseFactory {
public:
  Connection* createConnection() const override {
    return new MongoDBConnection>();
  }
};
```

```
class MySQLDBFactory : public DatabaseFactory {
public:
  Connection* createConnection() const override {
    return new MySqlConnection>();
  }
};
```

```
class Client {
  DatabaseFactory* factory;

public:
  Client(DatabaseFactory* f) : factory(f) {}

  void connect() {
    Connecton* connection =
        factory->createConnection();
    connection->connect();
  }
};
```

Coupling: Summary



- Low coupling is a general design goal
  - Simplifies understanding
  - Supports independent change and evolution
  - Allows reuse
- Coupling to stable classes is less critical
  - For example, using or inheriting from standard library classes
- Many design patterns reduce coupling – often at the cost of indirections

Modularity

- Coupling
- **Adaptation**
  - **Parameterization**
  - Specialization

Context

Previously

Coupling: an important *generic problem* that prevents modularity, and several possible solutions

Next

Adaptation: an important *generic approach* for achieving modularity (that can also reduce coupling)

Change

- Since software is (perceived as being) easy to change, software systems often deviate from their initial design

- Typical changes include
  - New features (requested by customers or management)
  - New interfaces (new hardware, new or changed interfaces to other software systems)
  - Bug fixing, performance tuning

- Changes often erode the structure of the system

→ Adaptation effectively means anticipating changes and preparing the code for (certain) changes

What can we make generic?

```
double sum (const std::vector<int>& v){
     double result = 0;
     for (auto x: v)
          result += x;
     return result;
 }
```

What can we make generic?

```
double sum (const std::vector<int>& v){
      double result = 0;
      for (auto x: v)
            result += x;
      return result;
  }
```

element type
(values)

container
(data structure)

initial value

elementary operation
(algorithm)

result type

How can we make it generic?

```
double sum (const std::vector<int>& v){
       double result = 0;
       for (auto x: v)
              result += x;
       return result;
  }
```

generic container / element type

generic result type

generic algorithm

```
template<type InputIt, type T, type BinaryOperation>
T accumulate(InputIt first, InputIt last, T init, BinaryOperation op) {
       T res = init;
       for (auto it = first; it != last; ++it)
           res = op(res, *it);
       return res;
}
```

generic data selection

Parameterization

- Modules can be prepared for change by allowing clients to influence their behavior

- Make modules *parametric* in:
  - The values they manipulate (e.g. integers `x, y, …`)
  - The types they operate on (e.g. `int, string, …`)
  - The data structures they operate on (e.g. `vector<int>, set<string>, …`)
  - The algorithms they apply (e.g. quicksort, radix sort, …)

Parameterization: Another Example

```
class Trender {
  std::istringstream& fst;
  std::istringstream& snd;

public:
  // constructor omitted for brevity

  int next() {
    int f, s;

    fst >> f;
    snd >> s;

    return std::max(f, s);
  }
};
```

Parameterization: Another Example

Source of data is a fixed class

Number of sources is fixed

Type of data is fixed

Filter criterion is fixed

```
class Trender {
  std::istringstream& fst;
  std::istringstream& snd;

public:
  // constructor omitted for brevity

  int next() {
    int f, s;

    fst >> f;
    snd >> s;

    return std::max(f, s);
  }
};
```

Parameterizing Values

```
class Trender {
  std::istringstream& fst;
  std::istringstream& snd;

public:
  int next() {
    int f, s;

    fst >> f;
    snd >> s;

    return std::max(f, s);
  }
};
```

Use variable values, instead of constant values (or variably many instead of constantly many)

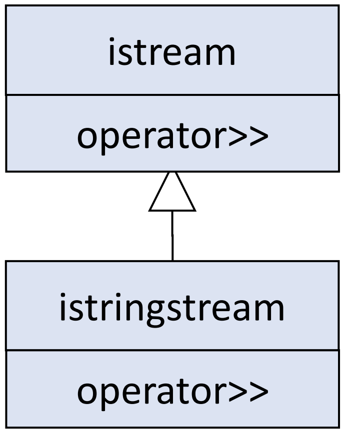
```
class Trender {
  std::vector<std::istringstream*> sources;

public:
  int next() {
    std::vector<int> values;

    // Read a value from each source
    for (auto src : sources) {
      int v;
      *src >> v;
      values.push_back(v);
    }

    return *std::max_element(values);
  }
};
```

Parameterizing Data Structures

```
class Trender {
  std::vector<std::istringstream*>
      sources;

public:
  int next() {
    std::vector<int> values;

    for (auto src : sources) {
      int v;
      *src >> v;
      values.push_back(v);
    }

    return *std::max_element(values);
  }
};
```

Use abstract classes (with inheritance or templates) to abstract over concrete implementations

```
class Trender {
  std::vector<std::istream*>
      sources;

public:
  int next() {
    // ... unchanged ...
  }
};
```



Parameterizing Types

```
class Trender {
  std::vector<std::istream*> sources;

public:
  int next() {
    std::vector<int> values;

    for (auto src : sources) {
      int v;
      *src >> v;
      values.push_back(v);
    }

    return *std::max_element(values);
  }
};
```

Use abstract types (with templates; or generics in, e.g. Java) to abstract over concrete types

```
template<typename E>
class Trender {
  std::vector<std::istream*> sources;

public:
  E next() {
    std::vector<E> values;

    for (auto src : sources) {
      E v;
      *src >> v;
      values.push_back(v);
    }

    return *std::max_element(values);
  }
};
```

Parameterizing Algorithms

```
template<typename E>
class Trender {
  std::vector<std::istream*> sources;

public:
  E next() {
    std::vector<E> values;

    for (auto src : sources) {
      E v;
      *src >> v;
      values.push_back(v);
    }

    return *std::max_element(values);
  }
};
```

Parameterize to abstract over concrete algorithms and behavior (e.g. via template metaprogramming)

```
template<typename E, typename SF>
class Trender {
  std::vector<std::istream*> sources;
  SF select_func;

public:
  E next() {
    // ... unchanged ...

    return select_func(values);
  }
};
```

```
Trender(
  ...,
  [](auto& vals) {
    return *std::min_element(vals.begin(), vals.end());
  });
```

See also the *strategy* pattern, if you are interested in a related pattern.

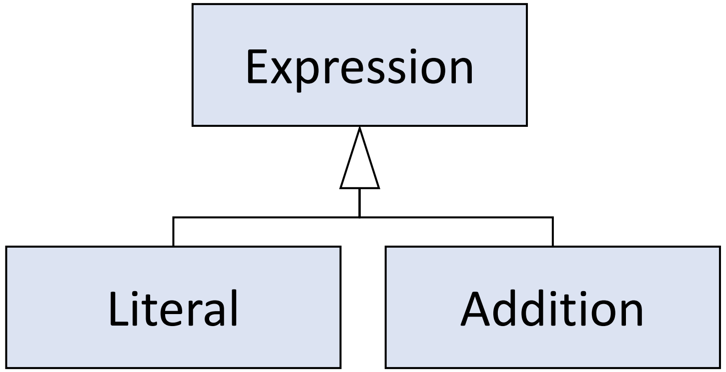
Getting this to work in C++ isn't trivial, see https://stackoverflow.com/questions/66762246

Modularity

- Coupling
- **Adaptation**
  - Parameterization
  - **Specialization**

Example: Operations Over Expressions

- Consider a data structure with elements of different types, e.g. expressions
- The behavior of operations on expressions, e.g. evaluation, depends on the type of expression it is applied to
- The **set of expressions is not fixed**

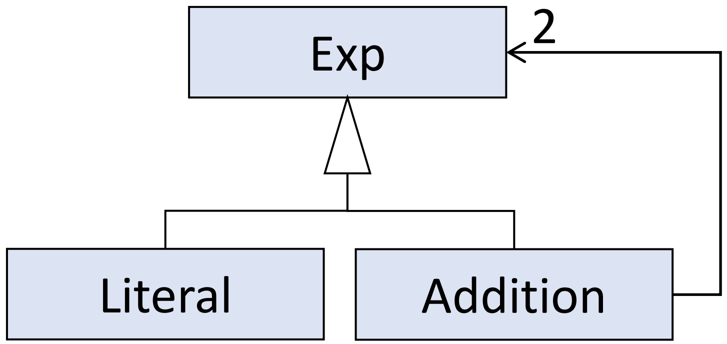


Operations

- Evaluate
- Prettyprint
- Simplify
- Derivate, integrate
- …

Dynamic Method Binding

- In object-oriented programs, behaviors can be specialized via overriding and dynamic method binding




```
class Exp {
  virtual double eval() const = 0
};
```

```
class Literal : public Exp {
  …
  Literal(double value): val(value) {}
  double eval() const { return val; }
};
```

```
class Addition : public Exp {
  …
  Addition(Exp* l, Exp* r): lhs(l), rhs(r) {}
  double eval() const {
    return lhs->eval() + rhs->eval();  }
};
```

Dynamic Method Binding as Case Distinction

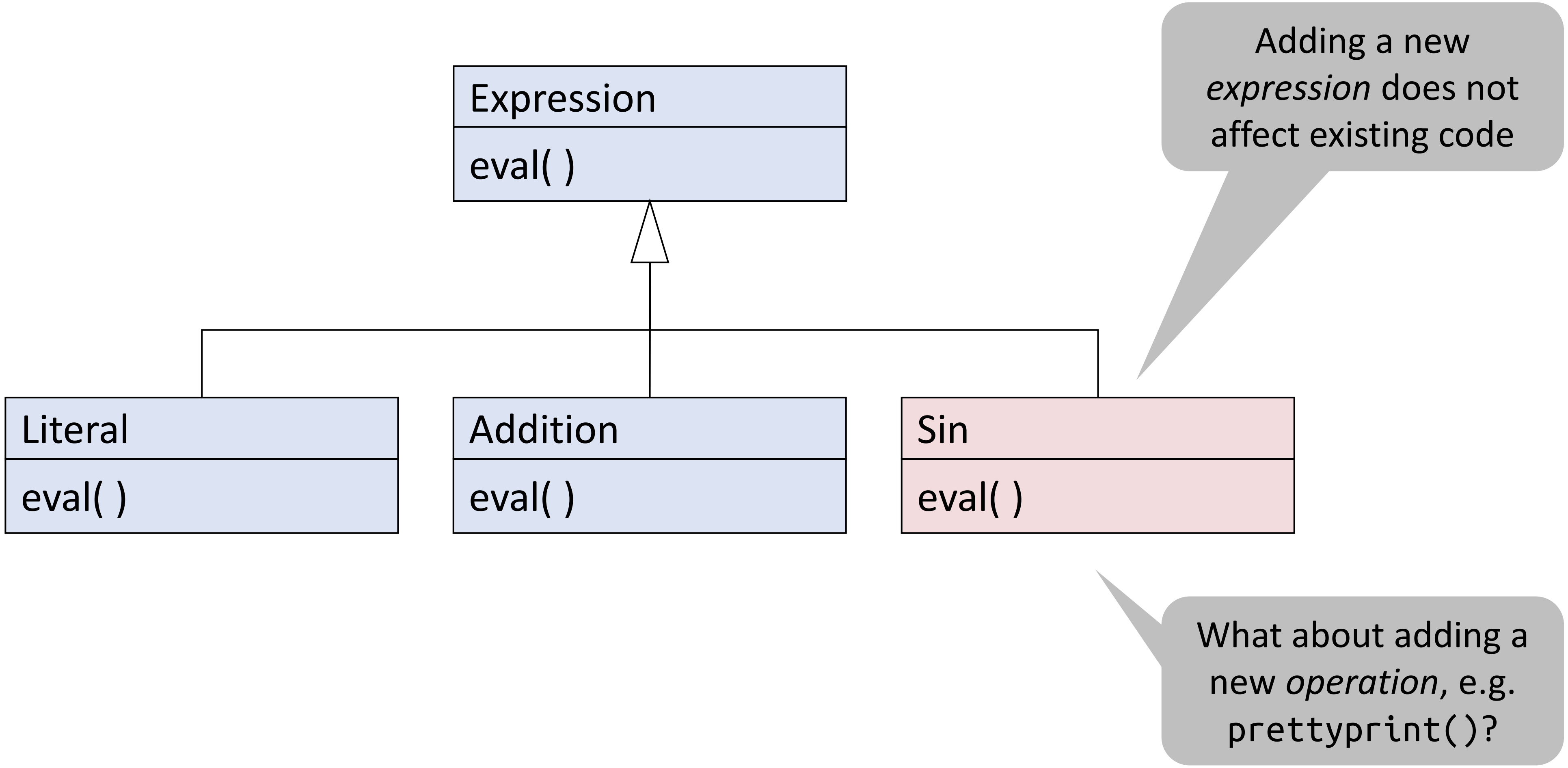
- **Dynamic method binding** is a case distinction on the dynamic type of the receiver object

```
s.eval();
```

```
// Pseudocode illustrating runtime behavior
if (s instanceof Addition)
  return Addition::eval(s);
else if (s instanceof Literal)
 return Literal::eval(s);
else if …
```

- Distinction is done by the language runtime behind the scenes
  - Adding or removing method overrides (cases) does not require changes outside of the overriding class
  - This makes existing code (other overrides, clients) adaptable

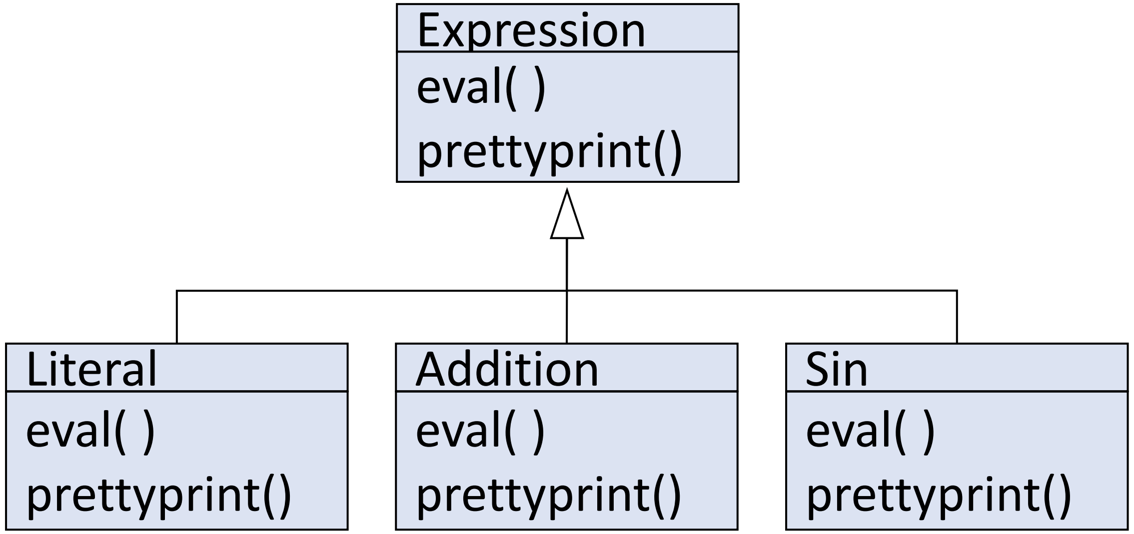
Adaptation: New Expressions



Interlude: Static vs. Dynamic Method Binding

- Dynamic method binding has drawbacks
  - Reasoning: Subclasses share responsibility for maintaining invariants
  - Testing: Dynamic binding increases the number of possible behaviors that need to be tested
  - Versioning: Dynamic binding makes it harder to evolve code without breaking subclasses
  - Performance: Overhead of method look-up at run-time
- Choose binding carefully for each method
  - Java: Consider making methods final
  - C++, C#: Consider making methods virtual (and final)

Example Reloaded

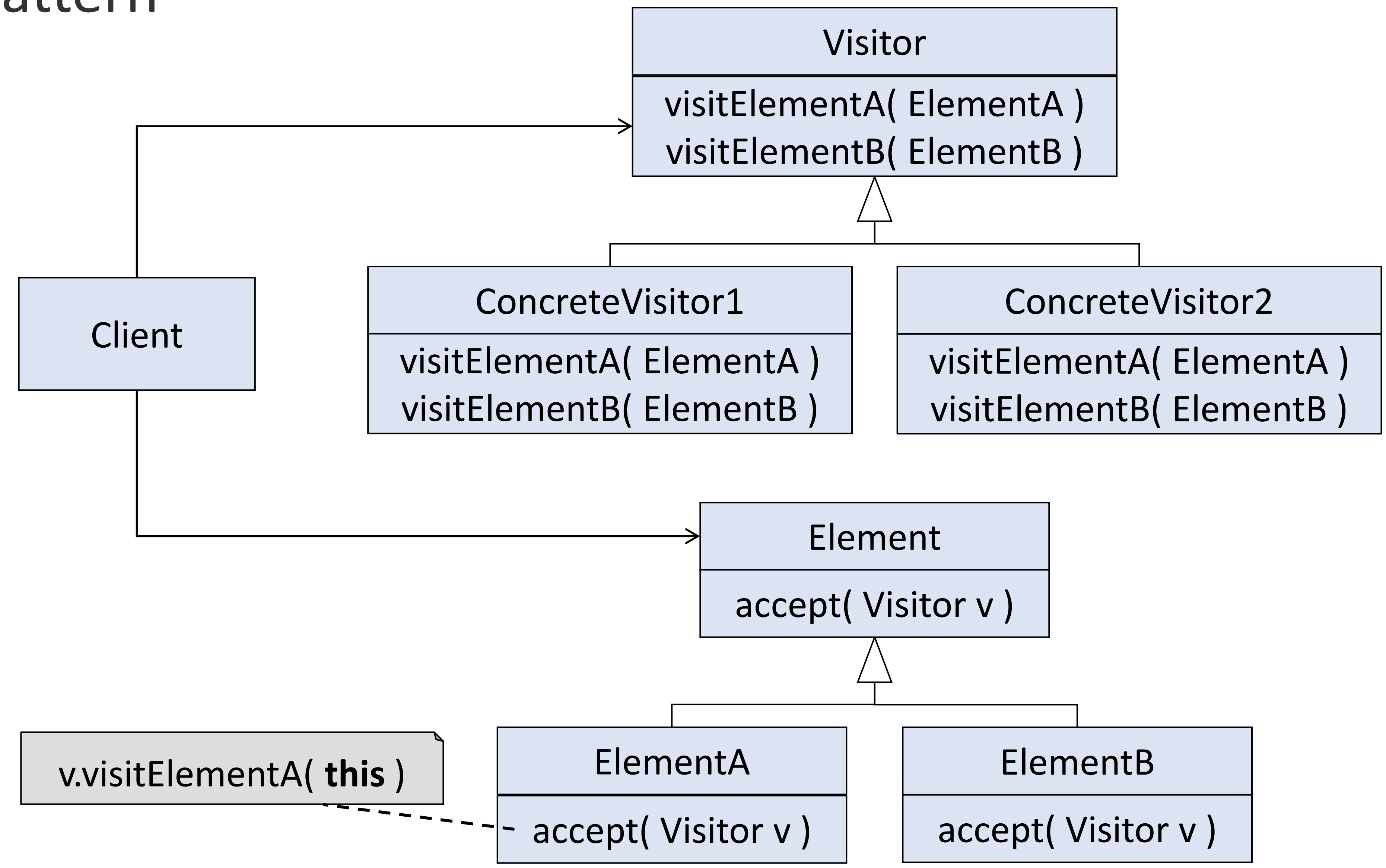


Expressions
- Cos
- Power
- Sqrt
- …

Operations
- Simplify
- Derivate
- Substitute
- …

- Seen: Dynamic binding enables adding new *operations*
- Can we find a similarly flexible solution that enables adding new *expressions* without having to change existing code?

Visitor Pattern



Visitor Pattern

```
class Exp {
  virtual void accept(Visitor)= 0;
};
```

```
class Literal : public Exp {
...
  void accept(Visitor v){
    v.visitLiteral(this)
  }
}
```

```
class Addition : public Exp {
  ...
  void accept(Visitor v) {
    v.visitAddition(this)
  }
}
```

Full example on Code Expert

```
class Visitor {
  virtual void visitLiteral(Literal e) = 0;
  virtual void visitAddition(Addition e) = 0;
}
```

```
class Evaluator extends Visitor {
  double value;

  void visitLiteral(Literal e) {
    value = e.value;
  }

  void visitAddition(Addition e) {
     Evaluator l; e.left.accept(l)
     Evaluator r; e.right.accept(r)
     value = l.value + r.value;
  }
}
```

```
class PrettyPrinter extends Visitor {
  …
}
```

Advantages and Drawbacks of the Visitor Pattern

- New behavior (operations) can be added to work with different classes without changing these classes
- A visitor can be used to accumulate useful information while working with various objects (not shown here)

- All visitors require updating when a class (e.g. a concrete expression) is added or removed
- Indirections complicate code reasoning
- The visitor functions have a fixed signature. If required, additional information that needs to be passed from visitor method to visitor method needs to go over the visitor (➜ additional state; can complicate recursion)

Adaptation: Summary

- Designing adaptable modules
  - Makes inevitable changes easier
  - Facilitates reuse

- Parameterization allows clients to customize the behavior by supplying different parameters

- Specialization allows clients to customize behavior by adding subclasses and overriding methods

- *But be careful: Making things abstract makes code harder to understand*