Structural Design Patterns

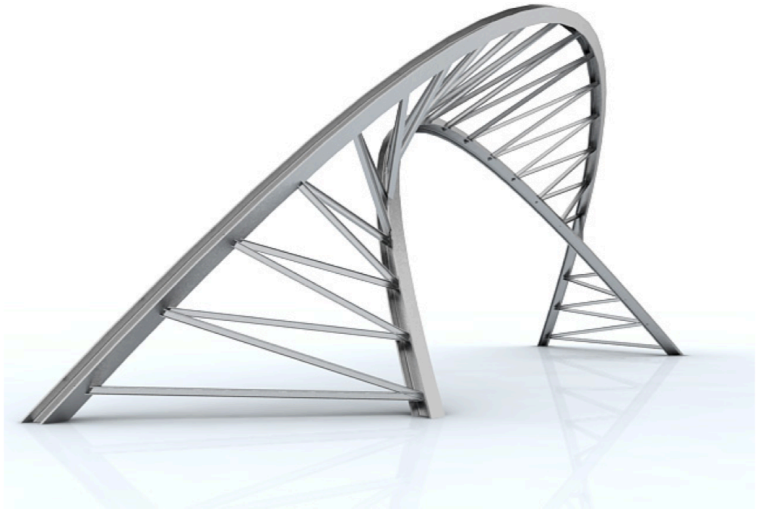
Structural Design Patterns

- Concerned with how classes/objects are composed to form larger structures

- Two types:
  - Class Structural
    - Use inheritance to compose interfaces and implementations
  - Object Structural
    - Describe ways to compose objects to realize new functionality
    - Added flexibility → can change composition at run-time

Structural Design Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Flyweight



Structural Patterns: Adapter

- Real-world example from

  http://stackoverflow.com/questions/3478225/when-do-we-need-adapter-pattern
- Suppose we are writing software to control DVRs made by various manufacturers
- All DVRs essentially have the same functionality:
  - Play, Pause, Rewind, etc.

Structural Patterns: Adapter

- Each DVR manufacturer provides a software library which allows code to control the DVR

- Problem: each library has a different interface:
  - Cisco: `beginPlayBack(Time startTime);`
  - Sony: `startPlayback(long seconds);`
  - Tivo: `playFrom(int h, int m, int s);`

Structural Patterns: Adapter

CiscoDvr.h

```
class CiscoDvr
{
  public:
    typedef struct
    {
      int hour;
      int minute;
      int second;
    } Time;

    void beginPlayBack(Time startTime);
    void stopPlayBack();
    void pausePlayBack();
};
```

Structural Patterns: Adapter

SonyDvr.h

```
class SonyDvr
{
  public:
    void startPlayback(long seconds);
    void stop();
    void pause();
};
```

Structural Patterns: Adapter

TivoDvr.h

```
class TivoDvr
{
  public:
    void playFrom(int h, int m, int s);
    void stopPlayback();
    void pausePlayback();
};
```

Structural Patterns: Adapter

```
//Control any type of DVR
if (type == "Cisco")
{
   Time t;
   t.hour = 1;
   t.minute = 48
   t.second = 23;

   CiscoDVR* dvr = new CiscoDVR();
   dvr->beginPlayBack(t);
}
else if (type == "Sony")
{
   SonyDVR* dvr = new SonyDVR();
   dvr->startPlayback(6503);
}
else if (type == "Tivo")
{
   TivoDVR* dvr = new TivoDVR();
   dvr->playFrom(1, 48, 23);
}
```

- Yuck!

Structural Patterns: Adapter

Design Pattern:

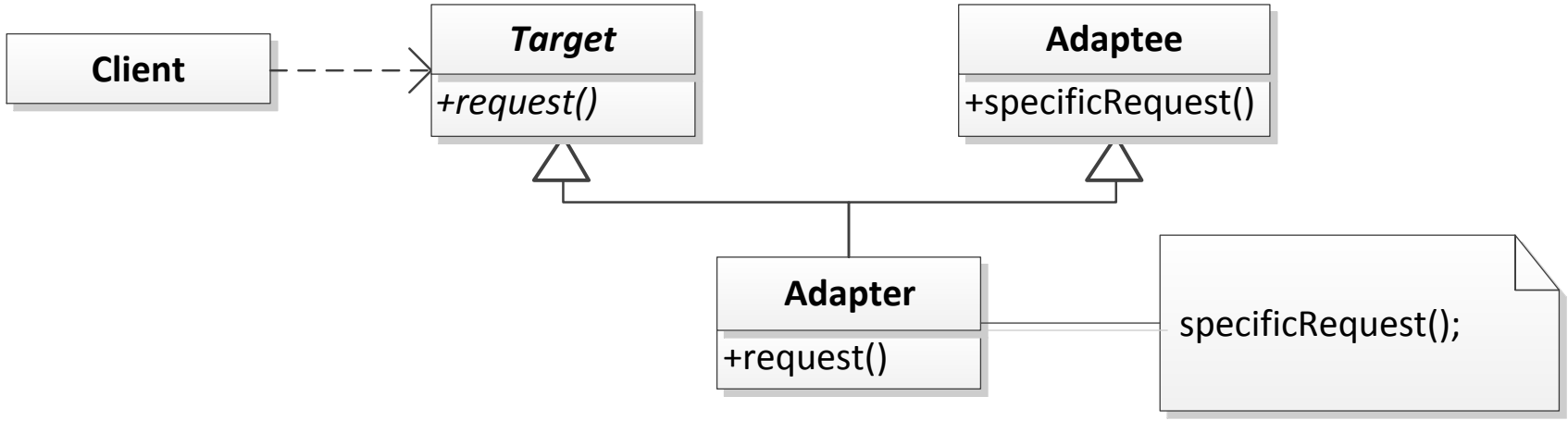
Adapter

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Structural Patterns: Adapter

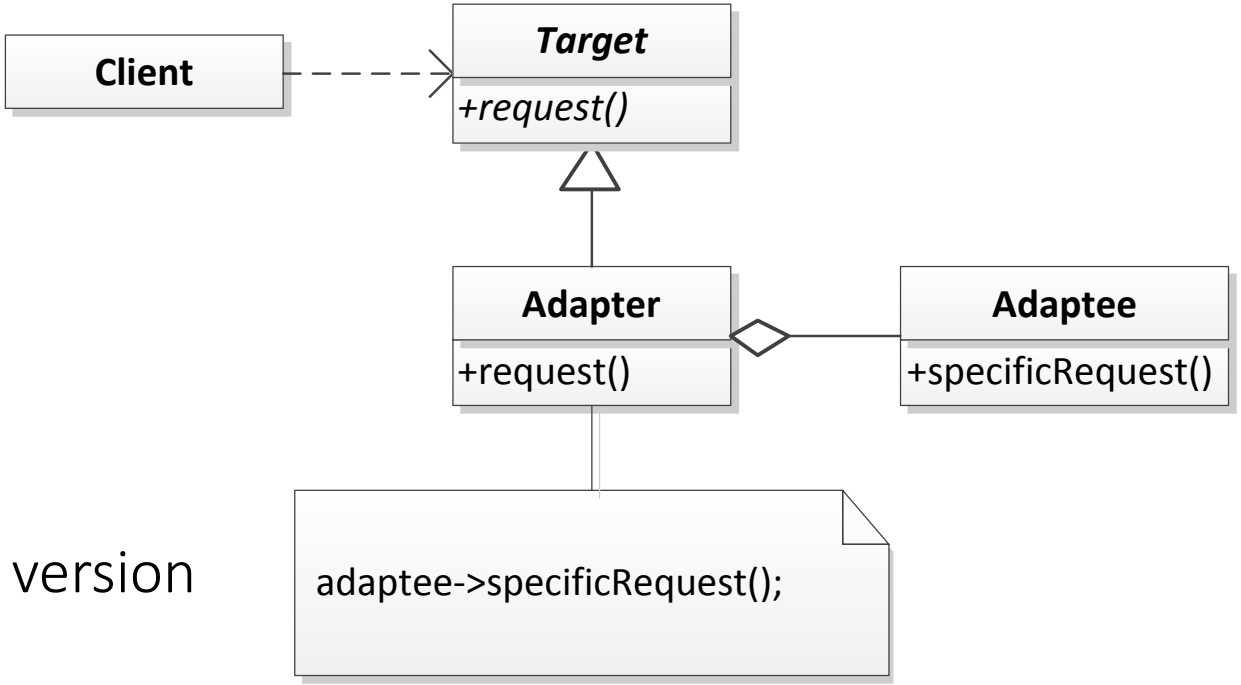
- Applicability:
  - You want to use an existing class, and its interface does not match the one you need
  - You want to create a reusable class that cooperates with unrelated or unforeseen classes that don’t necessarily have compatible interfaces
  - (Object Adapter only) You need to use several existing subclasses, but it’s impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Structural Patterns: Adapter



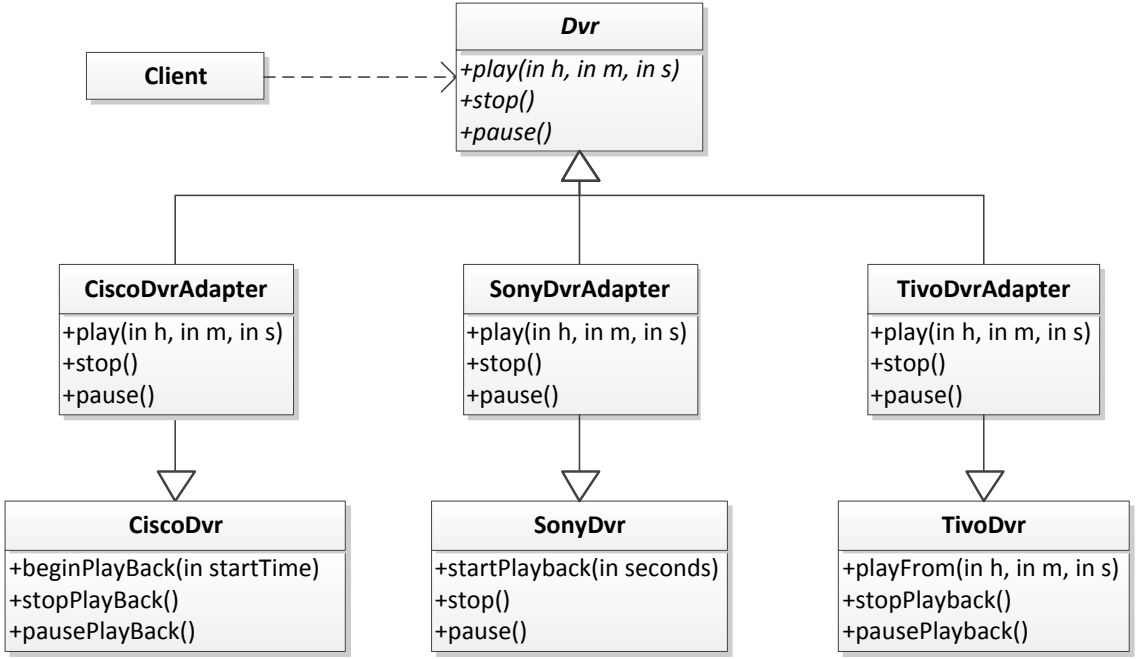
Class structural version

Structural Patterns: Adapter



Object structural version

Structural Patterns: Adapter



Structural Patterns: Adapter

CiscoDvrAdapter.h

```
class CiscoDvrAdapter : public Dvr, private CiscoDvr
{
  public:
    void play(int h, int m, int s);
    void stop();
    void pause();
};
```

SonyDvrAdapter.h

```
class SonyDvrAdapter : public Dvr, private SonyDvr
{
  public:
    void play(int h, int m, int s);
    void stop();
    void pause();
};
```

TivoDvrAdapter.h

```
class TivoDvrAdapter : public Dvr, private TivoDvr
{
  public:
    void play(int h, int m, int s);
    void stop();
    void pause();
};
```

Structural Patterns: Adapter

- Note the use of `private` and `public` inheritance in the adapter classes.

- Why?
  - Our adapters look and act as a `Dvr` and only need to make use of the device specific interfaces …

Structural Patterns: Adapter

CiscoDvrAdapter.cpp

```
void CiscoDvrAdapter::play(int h, int m, int s)
{
  Time t;

  t.hour = h;
  t.minute = m;
  t.second = s;

  this->beginPlayBack(t);
}

void CiscoDvrAdapter::stop()
{
  this->stopPlayBack();
}

void CiscoDvrAdapter::pause()
{
  this->pausePlayBack();
}
```

Structural Patterns: Adapter

```
void SonyDvrAdapter::play(int h, int m, int s)
{
  long seconds = (h * 3600) + (m * 60) + s;
  this->startPlayback(seconds);
}

void SonyDvrAdapter::stop()
{
  SonyDvr::stop();
}

void SonyDvrAdapter::pause()
{
  SonyDvr::pause();
}
```

- What's up with `SonyDvr::stop();` and `SonyDvr::pause();` ? We have a name collision between `SonyDvrAdapter` and `SonyDvr`, so `this->stop();` isn't specific enough …

Structural Patterns: Adapter

TivoDvrAdapter.cpp

```
void TivoDvrAdapter::play(int h, int m, int s)
{
  this->playFrom(h, m, s);
}

void TivoDvrAdapter::stop()
{
  this->stopPlayback();
}

void TivoDvrAdapter::pause()
{
  this->pausePlayback();
}
```

Structural Patterns: Adapter

main.cpp

```
void playEpisode(Dvr* dvr)
{
  dvr->play(0,45,10);
  dvr->pause();
  dvr->play(0,45,45);
  dvr->stop();
  cout << endl;
}

main()
{
  Dvr* dvr = new CiscoDvrAdapter();
  playEpisode(dvr);
  delete dvr;

  dvr = new SonyDvrAdapter();
  playEpisode(dvr);
  delete dvr;

  dvr = new TivoDvrAdapter();
  playEpisode(dvr);
  delete dvr;
}
```

Structural Patterns: Adapter

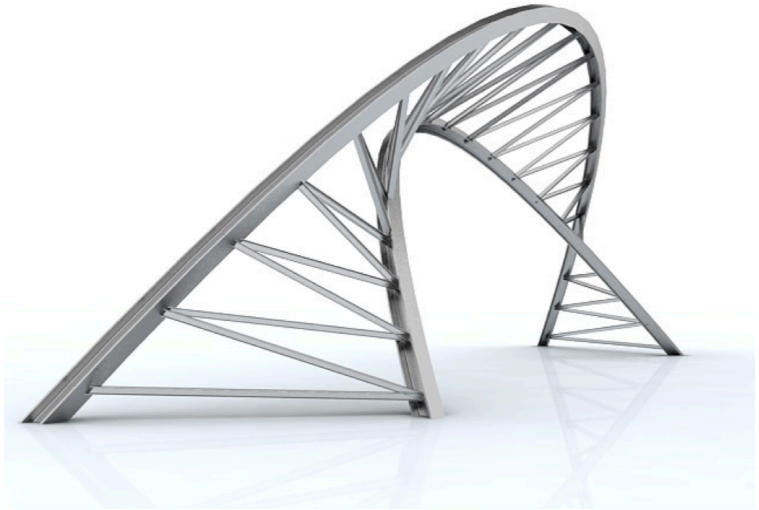
- Consequences (class adapter):
  - Adapts an Adaptee to Target by committing to a concrete Adapter class; a class adapter won't work when we want to adapt a class and all of its subclasses
  - Lets Adapter override some of Adaptee's behaviour, since it's a subclass of Adaptee
  - Introduces only one object, and no additional pointer indirection is needed to get to the adaptee

Structural Patterns: Adapter

- Consequences (object adapter):
  - Lets a single Adapter work with many Adaptees including both the Adaptee itself and all of its subclasses
  - Makes it harder to override Adaptee behaviour as it will require subclassing Adaptee and making Adapter refer to the subclass instead of the Adapter itself

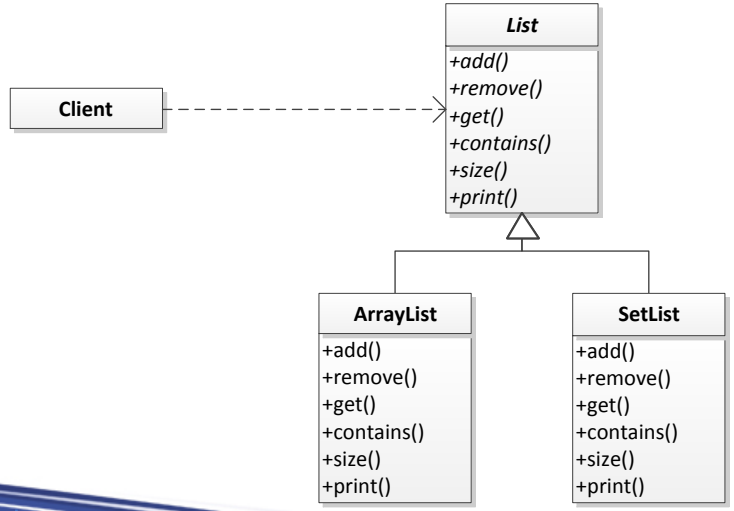
Structural Design Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Flyweight



Structural Patterns: Bridge

- Suppose we are implementing a list data structure
  - We wish to have multiple implementations: `ArrayList`, `SetList`, etc.

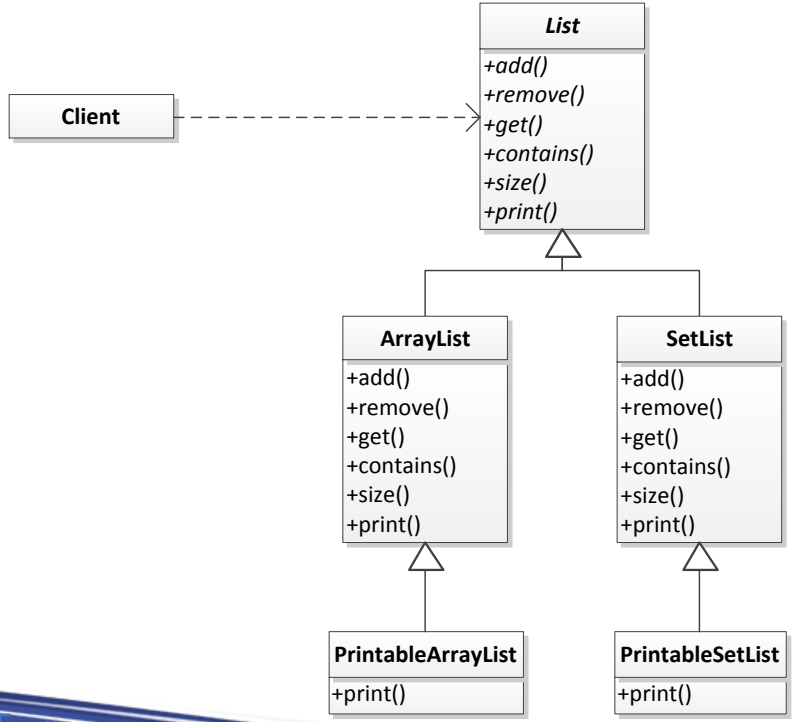


Structural Patterns: Bridge

- We later realize that we wish to add a refinement to the list abstraction:
  - `PrintableList` : Will override the `print` method to print the list in a nice way

```
Output

---------------------------
| 0 | 55 | 67 | 68 | 95 |
---------------------------
```

Structural Patterns: Bridge



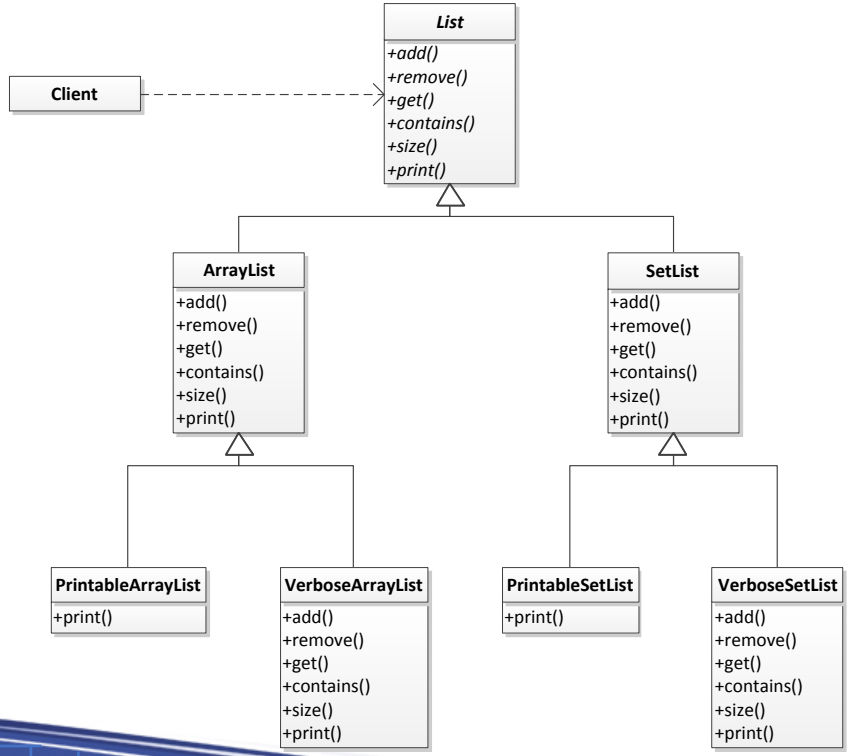
Structural Patterns: Bridge

- We later realize that we wish to add another refinement to the list abstraction:
  - `VerboseList`: Used for debugging. Prints a message each time one of its methods is called.

`Output`

```
Adding value 55
Retrieving list size
Printing list contents
[0]: 67 [1]: 68 [2]: 0 [3]: 95 [4]: 55 [5]: 55

Retrieving element at index 0
Removing value 55
Retrieving list size
```

Structural Patterns: Bridge



Structural Patterns: Bridge

- Problem:
  - Each time we add a refinement, we have to create a new subclass for each list implementation class
  - Lots of code will be duplicated
  - The inheritance hierarchy will become quite complex
- If only we knew of a way to somehow separate the list abstraction from the list implementation...

Structural Patterns: Bridge

Design Pattern:

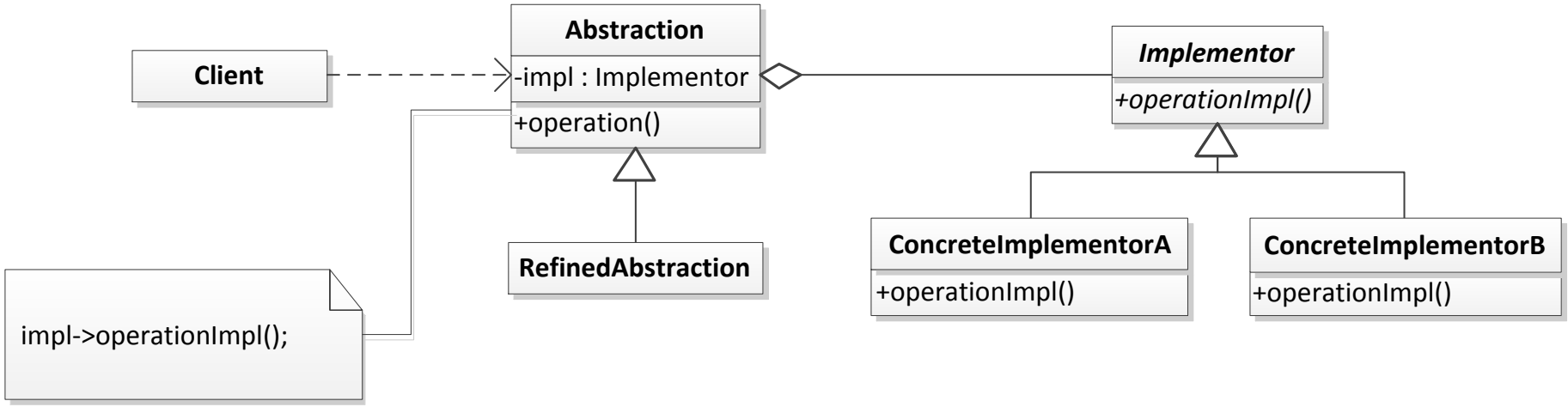
Bridge

Decouple an abstraction from its implementation so that the two can vary independently.

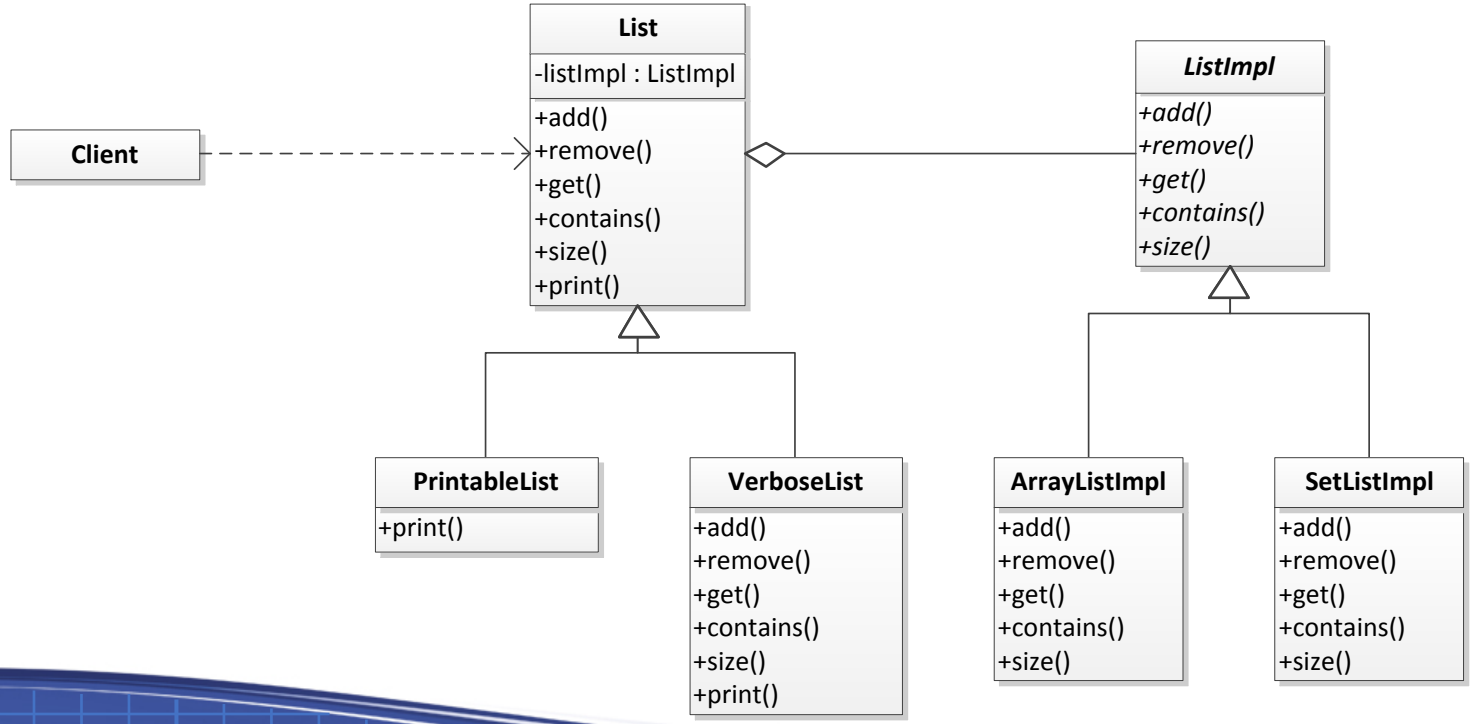
Structural Patterns: Bridge

- Applicability:
  - You want to avoid a permanent binding between an abstraction and its implementation
  - Both the abstractions and their implementations should be extensible by subclassing; Bridge lets you combine the different abstractions and implementations and extend them independently
  - Changes in the implementation of an abstraction should have no impact on clients
  - You want to share an implementation among multiple objects and this fact should be hidden from the client

Structural Patterns: Bridge



Structural Patterns: Bridge



Structural Patterns: Bridge

List.h

```
template <class T>
class List
{
  public:
    List(ListImpl<T>* impl) : _listImpl(impl)
    {
    }

    virtual ~List()
    {
      delete this->_listImpl;
    }

    virtual void add(T value)
    {
      this->_listImpl->add(value);
    }

    virtual bool remove(T value)
    {
      return this->_listImpl->remove(value);
    }
```

Structural Patterns: Bridge

PrintableList.h

```
template <class T>
class PrintableList : public List<T>
{
  public:
    PrintableList(ListImpl<T>* impl) : List<T>(impl)
    {
    }

    virtual void print()
    {
      int borderLength = 0;
      std::ostringstream elements;

      for (int i = 0; i < this->_listImpl->size(); ++i)
      {
        std::ostringstream s;
        T value = this->_listImpl->get(i);

        s << value;
        borderLength += s.str().length() + 3;

        elements << "| " << value << " ";
      }
```

Structural Patterns: Bridge

VerboseList.h

```
class VerboseList : public List<T>
{
  public:
    VerboseList(ListImpl<T>* impl) : List<T>(impl)
    {
    }

    virtual void add(T value)
    {
      std::cout << "Adding value " << value << std::endl;
      List<T>::add(value);
    }

    virtual bool remove(T value)
    {
      std::cout << "Removing value " << value << std::endl;
      return List<T>::remove(value);
    }

    virtual bool contains(T value)
    {
      std::cout << "Checking if list contains value " << value << std::endl;
      return List<T>::contains(value);
    }
```

Structural Patterns: Bridge

ListImpl.h

```
template <class T>
class ListImpl
{
  public:
    virtual ~ListImpl()
    {
    }

    virtual void add(T value) = 0;
    virtual bool remove(T value) = 0;
    virtual bool contains(T value) = 0;
    virtual T get(int idx) = 0;
    virtual int size() = 0;
};
```

Structural Patterns: Bridge

ArrayListImpl.h

```
template <class T>
class ArrayListImpl : public ListImpl<T>
{
  public:
    ArrayListImpl()
    {
      this->_elements = new T[5];
      this->_capacity = 5;
      this->_nextIndex = 0;
    }

    virtual ~ArrayListImpl()
    {
      delete this->_elements;
    }

    virtual void add(T value)
    {
      if (_nextIndex == _capacity)
      {
        _capacity *= 2;
```

Structural Patterns: Bridge

SetListImpl.h

```
template <class T>
class SetListImpl : public ListImpl<T>, private std::set<T>
{
  public:
    virtual void add(T value)
    {
      this->insert(value);
    }

    virtual bool remove(T value)
    {
      return (this->erase(value) > 0);
    }

    virtual bool contains(T value)
    {
      return (this->find(value) != this->end());
    }
```

Structural Patterns: Bridge

main.cpp

```
list = new PrintableList<int>(new ArrayListImpl<int>());
printHeader("Testing printable list with array implementation");
testList(list, values);

delete list;

list = new PrintableList<int>(new SetListImpl<int>());
printHeader("Testing printable list with set implementation");
testList(list, values);

delete list;
```

Structural Patterns: Bridge

```
Output
==================================================
Testing printable list with array implementation
==================================================
----------------------------
| 7 | 61 | 57 | 3 | 54 | 54 |
----------------------------
==================================================
Testing printable list with set implementation
==================================================
------------------------
| 3 | 7 | 54 | 57 | 61 |
------------------------
```

Structural Patterns: Bridge

main.cpp

```
list = new VerboseList<int>(new ArrayListImpl<int>());
printHeader("Testing verbose list with array implementation");
testList(list, values);

delete list;

list = new VerboseList<int>(new SetListImpl<int>());
printHeader("Testing verbose list with set implementation");
testList(list, values);

delete list;
```

Structural Patterns: Bridge

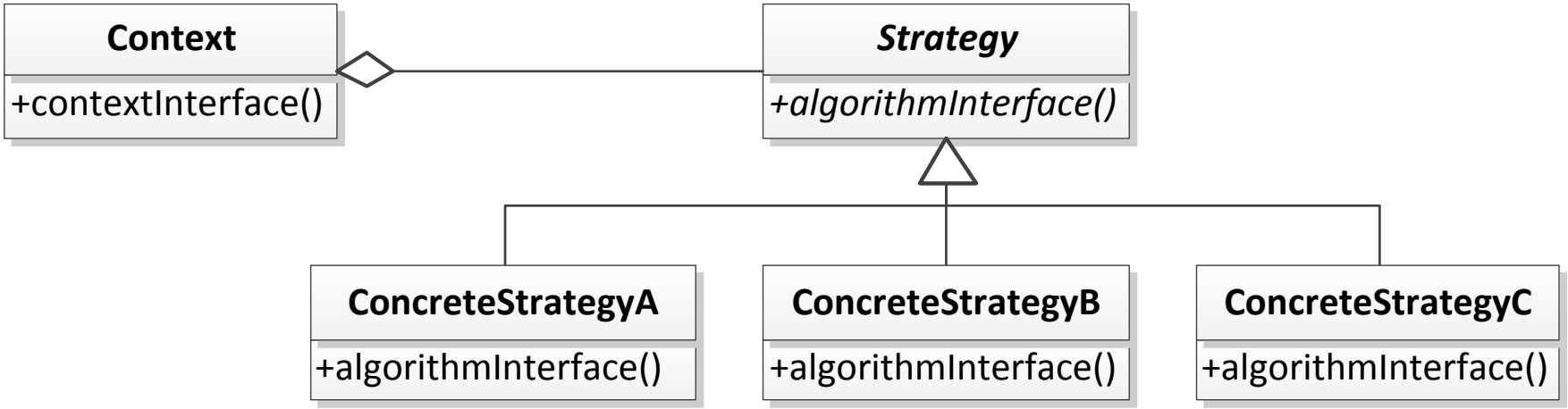
```
Output

================================================

Testing verbose list with array implementation

================================================

Adding value 79
Adding value 45
Adding value 53
Adding value 57
Adding value 44
Adding value 44
Retrieving list size
Printing list contents
[0]: 79 [1]: 45 [2]: 53 [3]: 57 [4]: 44 [5]: 44

================================================

Testing verbose list with set implementation

================================================

Adding value 79
Adding value 45
Adding value 53
Adding value 57
Adding value 44
Adding value 44
Retrieving list size
Printing list contents
[0]: 44 [1]: 45 [2]: 53 [3]: 57 [4]: 79
```

Structural Patterns: Bridge

- Consequences:
  - Decouples interface and implementation → the two are not bound permanently
  - Can potentially change implementation at run-time
  - Improved extensibility → can extend Abstraction and Implementor independently
  - Hides implementation details from clients

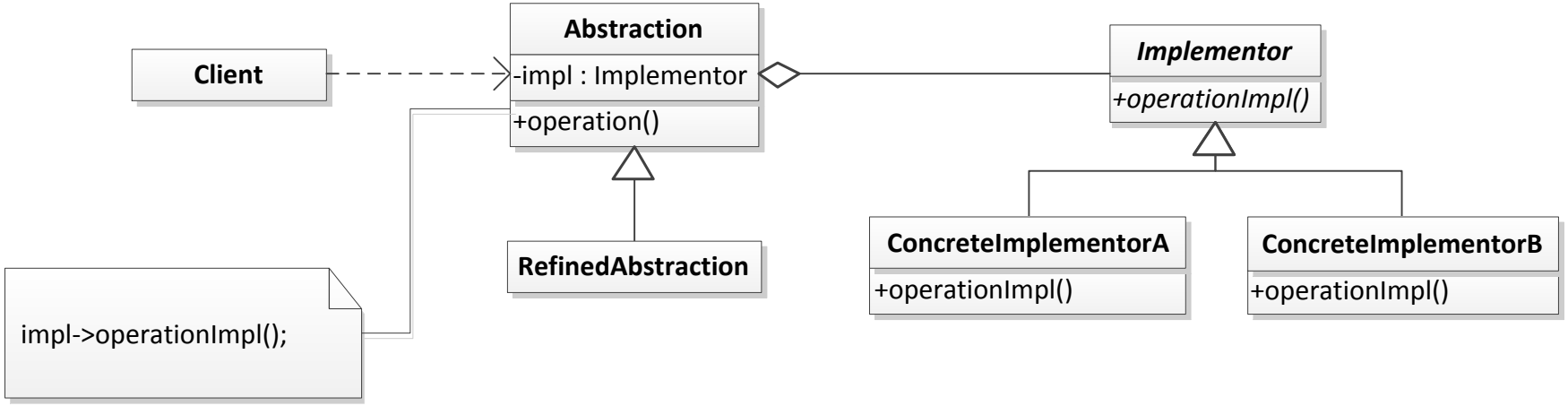
Structural Patterns: Bridge

- Recall the Strategy pattern:



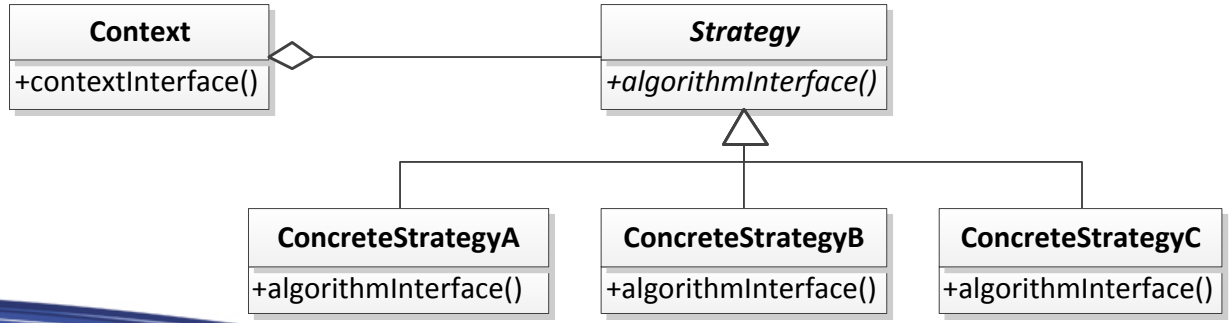
Structural Patterns: Bridge

- Notice any similarity?



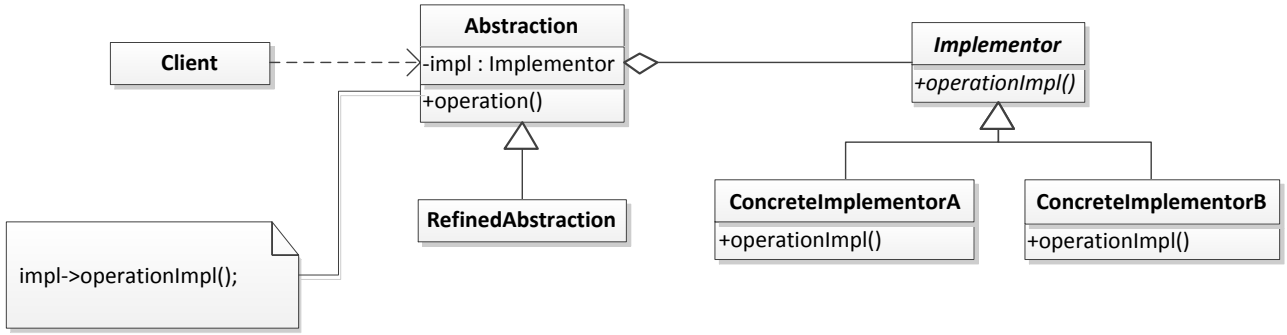
Structural Patterns: Bridge

- Intent is different
- Strategy:
  - Behavioural pattern concerned with providing interchangeable families of algorithms
  - Strategies may have little to no data stored within them
  - Want run-time flexibility on the lhs to change behaviour



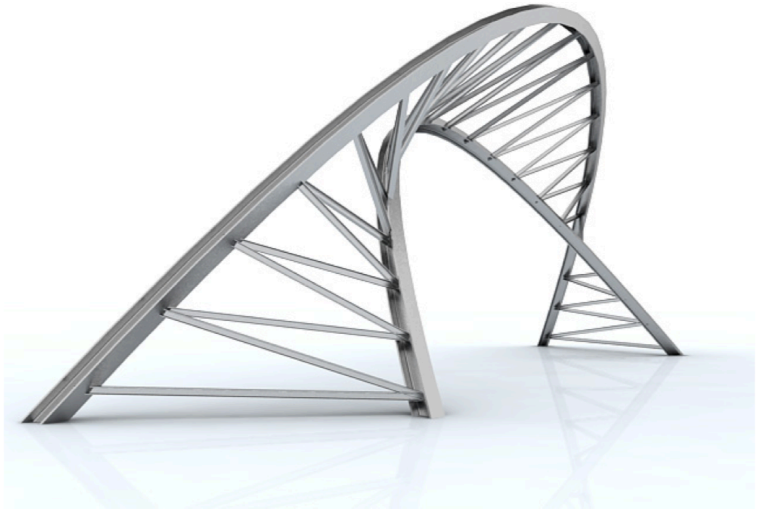
Structural Patterns: Bridge

- Bridge:
  - Structural pattern concerned with decoupling abstraction from implementation
    - Separation of interface and implementation
  - Want run-time flexibility on both the lhs and rhs



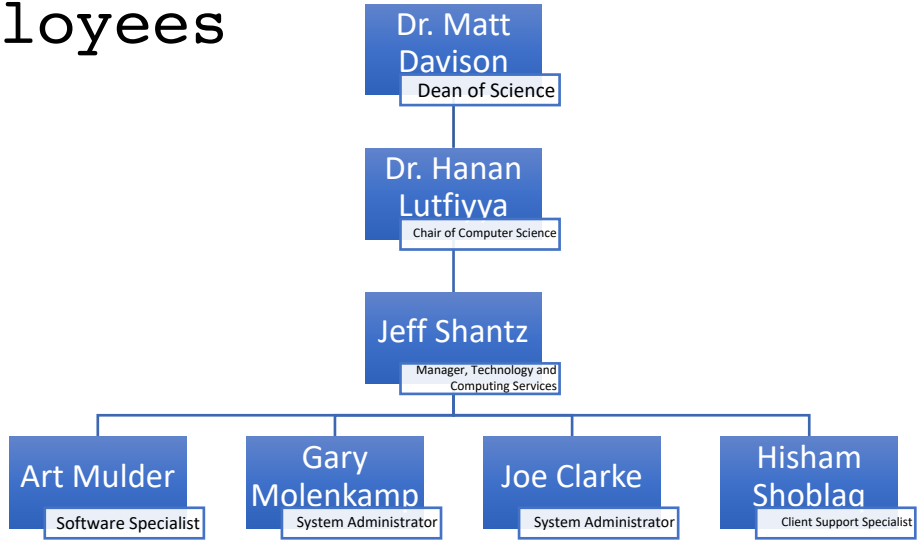
Structural Design Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Flyweight



Structural Patterns: Composite

- We wish to represent the Systems Group organization chart
  - Tree structure consisting of `Managers` and `RegularEmployees`



Structural Patterns: Composite

- We don't want clients to have to distinguish between a `Manager` and a `RegularEmployee`
  - e.g. we might have a `print` method:
    - `Manager::print` – Print the details of the manager and his/her subordinates
    - `RegularEmployee::print` – Print the details of the employee
  - Either way, the client shouldn't care whether it is dealing with a `Manager` or a `RegularEmployee`

Structural Patterns: Composite

Design Pattern:

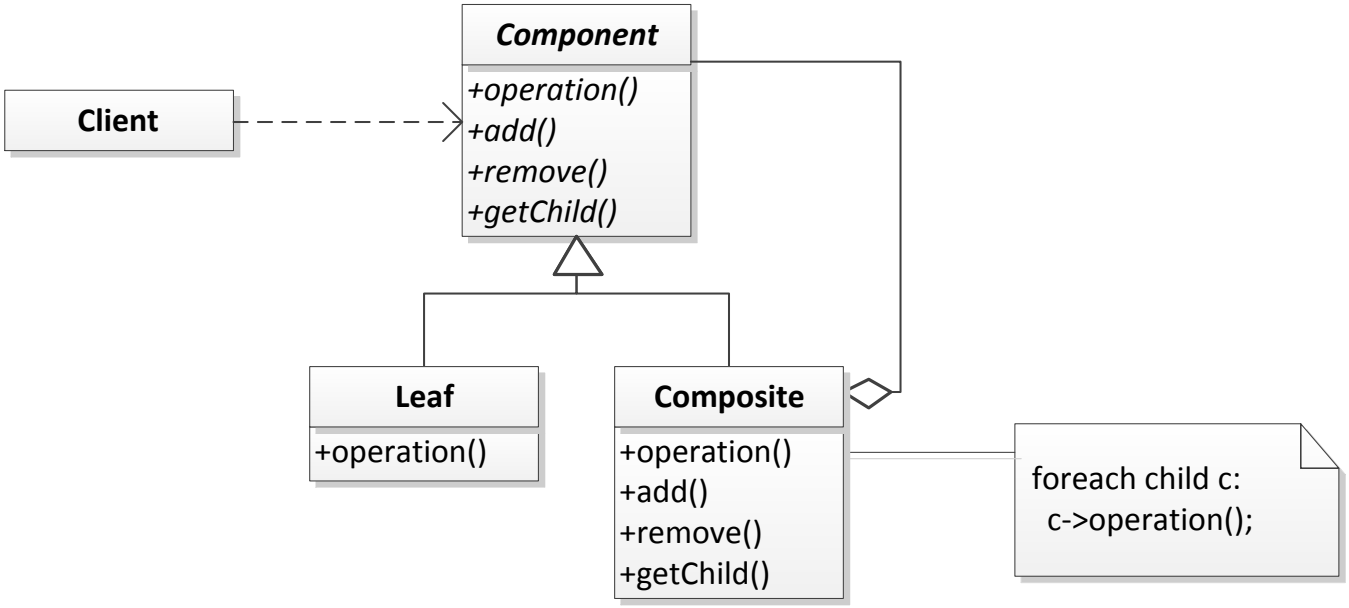
Composite

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

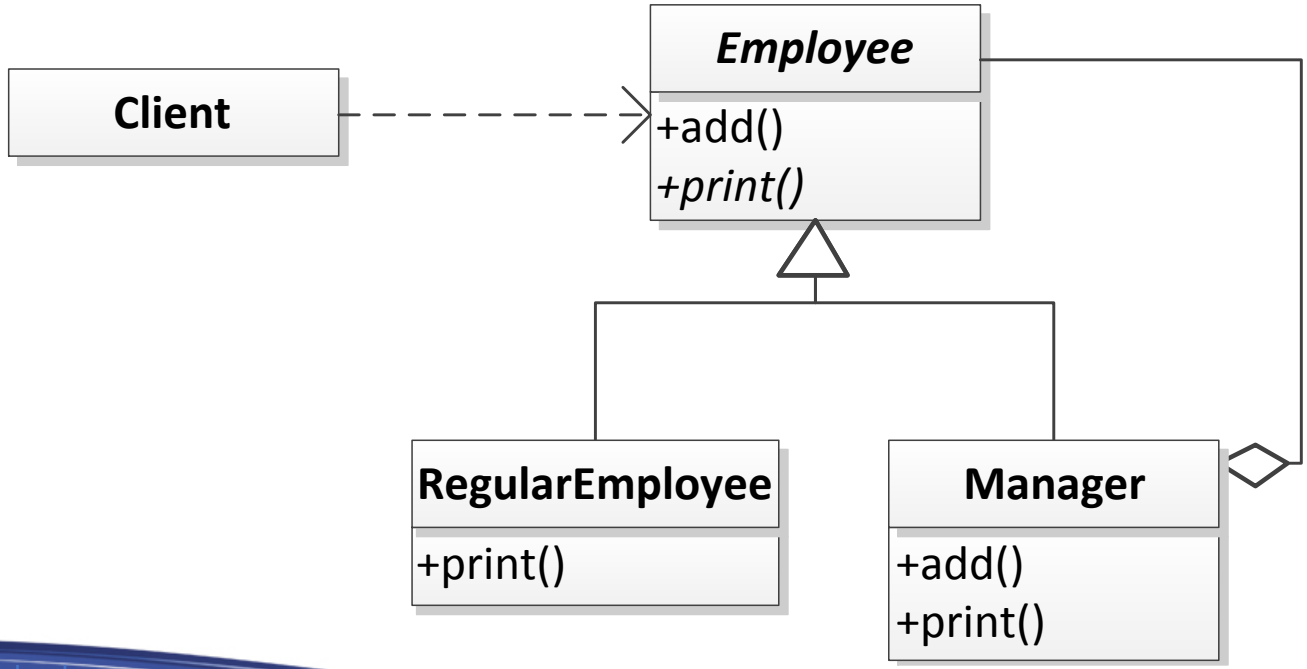
Structural Patterns: Composite

- Applicability:
  - You want to represent part-whole hierarchies of objects
  - You want clients to be able to ignore the difference between compositions of objects and individual objects; clients will treat all objects in the composite structure uniformly

Structural Patterns: Composite



Structural Patterns: Composite



Structural Patterns: Composite

Employee.h

```
class Employee
{
  public:
    Employee(const std::string& name) : _name(name)
    {
    }
    virtual ~Employee()
    {
    }
    virtual void add(Employee* member)
    {
      // do nothing -- throw exception?
    }
    virtual void print(int indent = 0) = 0;
    virtual const std::string& name() const
    {
      return this->_name;
    }
  private:
    std::string _name;
};
```

Structural Patterns: Composite

Manager.h

```
class Manager : public Employee
{
  public:
    Manager(const std::string& name);

    void add(Employee* member);
    void print(int indent);

  private:
    std::vector<Employee*> _subordinates;
};
```

Structural Patterns: Composite

RegularEmployee.h

```
class RegularEmployee : public Employee
{
  public:
    RegularEmployee(const std::string& name);
    void print(int indent);
};
```

Structural Patterns: Composite

main.cpp

```
main()
{
  Employee* dean = new Manager(”Matt Davison");
  Employee* chair = new Manager("Hanan Lutfiyya");
  Employee* manager= new Manager("Jeff Shantz");
  Employee* emp1 = new RegularEmployee("Art Mulder");
  Employee* emp2 = new RegularEmployee("Gary Molenkamp");
  Employee* emp3 = new RegularEmployee("Joe Clarke");
  Employee* emp4 = new RegularEmployee("Hisham Shoblaq");
  dean->add(chair);
  chair->add(manager);
  manager->add(emp1);
  manager->add(emp2);
  manager->add(emp3);
  manager->add(emp4);
  dean->print();
}
```

Structural Patterns: Composite

Output

```
Manager: Matt Davison
   Manager: Hanan Lutfiyya
      Manager: Jeff Shantz
         Employee: Art Mulder
         Employee: Gary Molenkamp
         Employee: Joe Clarke
         Employee: Hisham Shoblaq
```

Structural Patterns: Composite

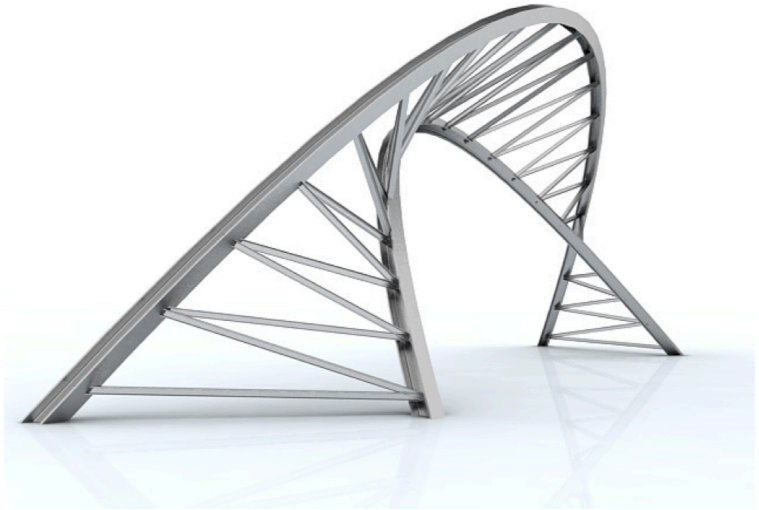
- Consequences:
  - Defines class hierarchies of primitive and composite objects
    - Primitive objects can be composed into more complex objects, which, in turn, can be composed
    - Wherever client code expects a primitive object, it can also take a composite object
  - Makes client code simple – doesn't have to distinguish between primitive and composite objects
  - Makes it easier to add new kinds of components
    - Newly-defined Leaf/Composite subclasses work automatically with existing structures and client code

Structural Patterns: Composite

- Consequences:
  - Can make the code overly general
    - Hard to restrict the components of a composite
- Note: Often used to model files and directories

Structural Design Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Flyweight



Structural Patterns: Decorator

- We often need to add functionality to an existing class
- Example from Java: `InputStreamReader`
  - Can read from any type of stream: file, character array, string, network connection
  - `int read()`
- For convenience, we would like to add buffering
  - `String readLine()`
  - `readLine` will make use of `read` and keep reading until it encounters a newline character

Structural Patterns: Decorator

- We refuse to modify `InputStreamReader`
  - Open / Closed Principle violations are bad
- Potential solution: subclass `InputStreamReader`
  - What about `FileReader`, `CharArrayReader`, `PushbackReader`, `PipedReader`, `StringReader` ?
    - Wouldn't we like to read a line from each of these readers as well?
    - Are we really going to subclass each and every one to provide buffering?
- If only we knew of a better way…

Structural Patterns: Decorator

Design Pattern:

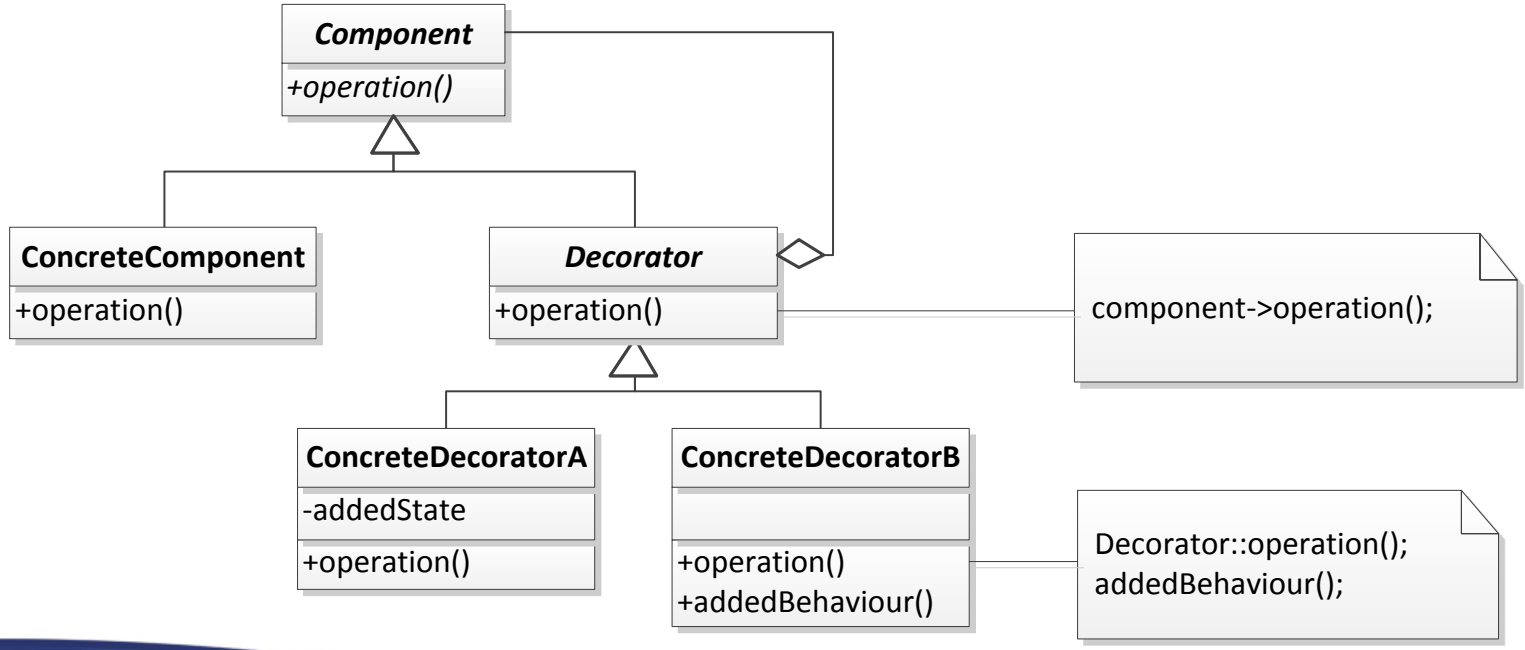
Decorator

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

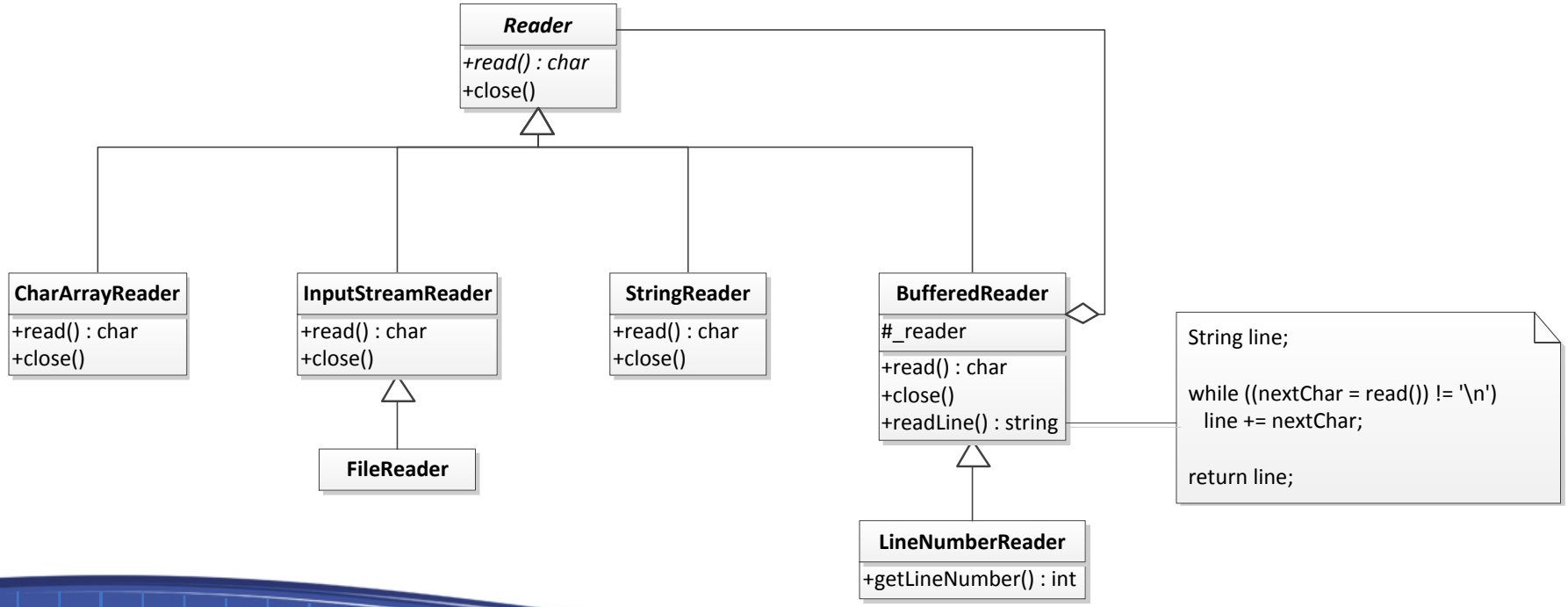
Structural Patterns: Decorator

- Applicability:
  - To add responsibilities to individual objects dynamically without affecting other objects
  - For responsibilities that can be withdrawn
  - When extension by subclassing is impractical
    - We may have a large number of independent extensions possible
    - The use of subclassing would produce an explosion of subclasses to support every combination

Structural Patterns: Decorator



Structural Patterns: Decorator



Structural Patterns: Decorator

- Another example from Java: input streams

```
FileInputStream fis = new FileInputStream("file.gz");
BufferedInputStream bis = new BufferedInputStream(fis);
GZIPInputStream gis = new GZIPInputStream(bis);
```

Structural Patterns: Decorator

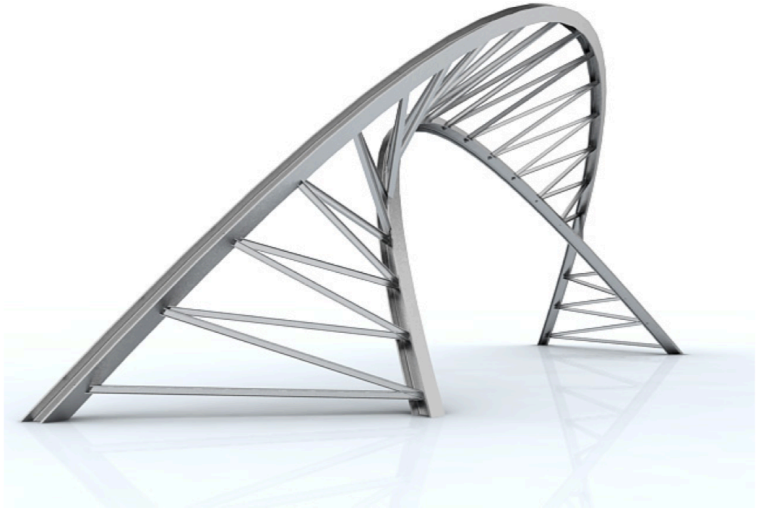
- Consequences:
  - Provides more flexibility than static inheritance
    - Can add/remove responsibilities at run-time by attaching/detaching decorators
  - Avoids feature-laden classes high up in the hierarchy
    - It would be inappropriate to put GZIP compression/decompression routines in the InputStream class
  - A decorator and its component are not identical
    - Decorators act as transparent enclosures
    - However, we can no longer rely on object identity when using decorators

Structural Patterns: Decorator

- Consequences:
  - Added complexity
    - Lots of little objects
    - Those new to Java often experience a WTF? moment when first discovering the many stream and reader classes available
    - “What happened to simple file I/O?”

Structural Design Patterns

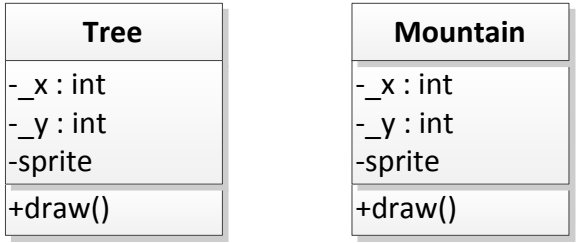
- Adapter
- Bridge
- Composite
- Decorator
- Flyweight



Structural Patterns: Flyweight

- Suppose we are creating a role-playing game
- We might have hundreds/thousands/millions of terrain objects:
  - `Tree`, `Mountain`, `Sea`, etc.

Structural Patterns: Flyweight



Suppose we need millions of these terrain objects

Problems:

- The cost alone of storing millions of objects in memory might be prohibitive
- Now multiply the millions of objects by the size of their sprites (images)
    - Sprites will usually be the same
    - `Tree` objects will have a sprite depicting a tree
    - `Mountain` objects will have a sprite depicting a mountain

Structural Patterns: Flyweight

Design Pattern:

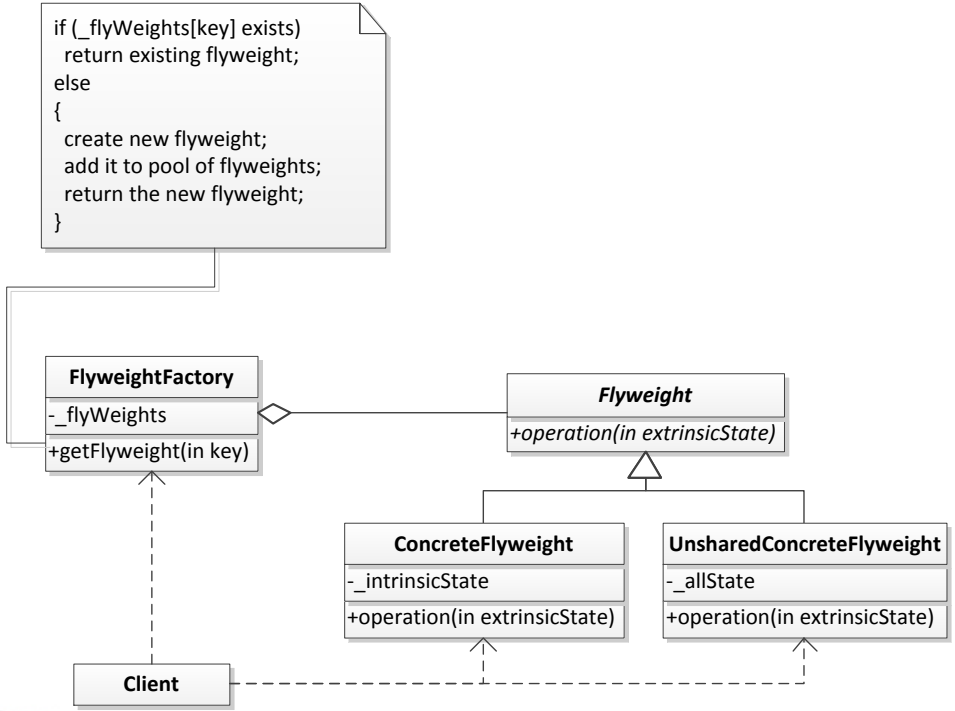
Flyweight

Use sharing to support large numbers of fire-grained objects efficiently.

Structural Patterns: Flyweight

- Applicability:
  - An application uses a large number of objects
  - Storage costs are high because of the sheer quantity of objects
  - Most object state can be made extrinsic
  - Many groups of objects can be replaced by relatively few shared objects once extrinsic state is removed
  - The application doesn’t depend on object identity
    - Since flyweight objects are shared, identity tests will return true for conceptually distinct objects

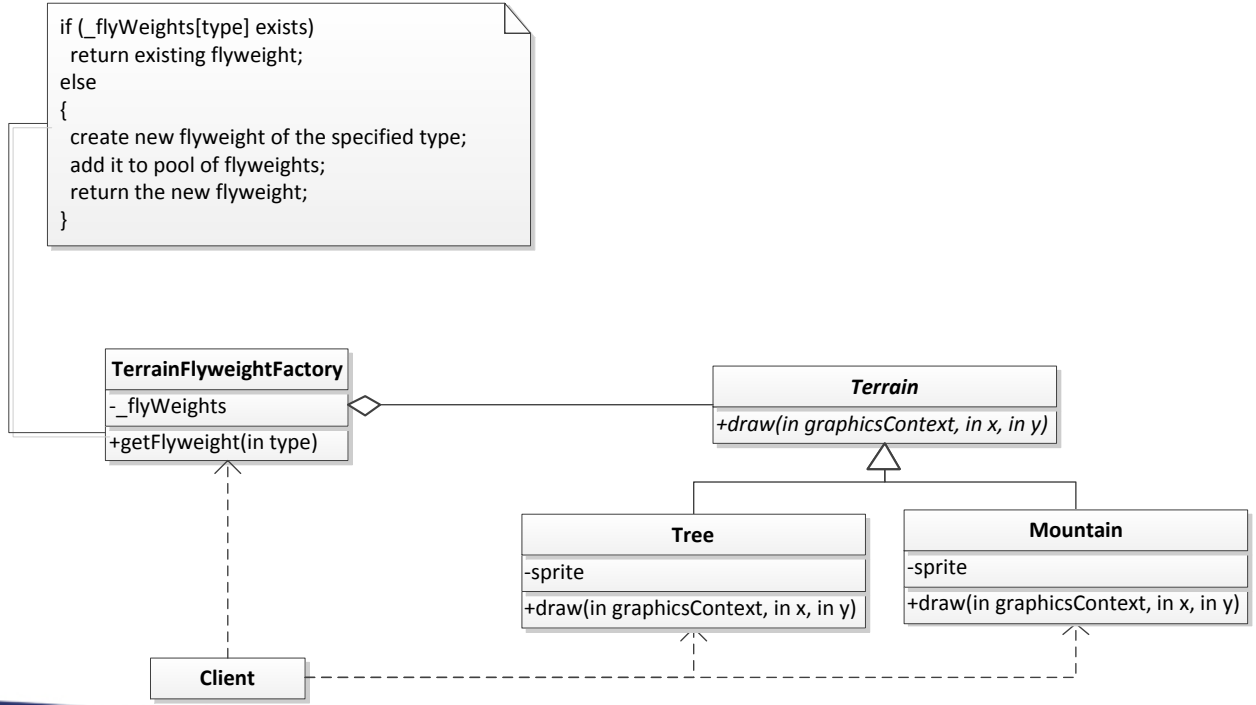
Structural Patterns: Flyweight



Structural Patterns: Flyweight

- Definitions:
  - Flyweight
    - A shared object that can be used in multiple contexts simultaneously
    - Acts as an independent object in each context – indistinguishable from an instance of the object that's not shared
  - Intrinsic state
    - Stored in the flyweight
    - Information that is independent of the flyweight's context, thus making it shareable
  - Extrinsic state
    - Depends on / varies with the flyweight's context
    - Can't be shared

Structural Patterns: Flyweight



Structural Patterns: Flyweight

- Instead of:

```
GraphicsContext* gc = getGraphicsContext();

for (int x = 0; x < 1000; ++x)
{
   for (int y = 0; y < 1000; ++y)
   {
      Tree* t = new Tree(gc, "tree.png", x, y);
      t->draw();
   }
}
```

- 1 million objects created
- `tree.png` loaded and stored 1 million times

Structural Patterns: Flyweight

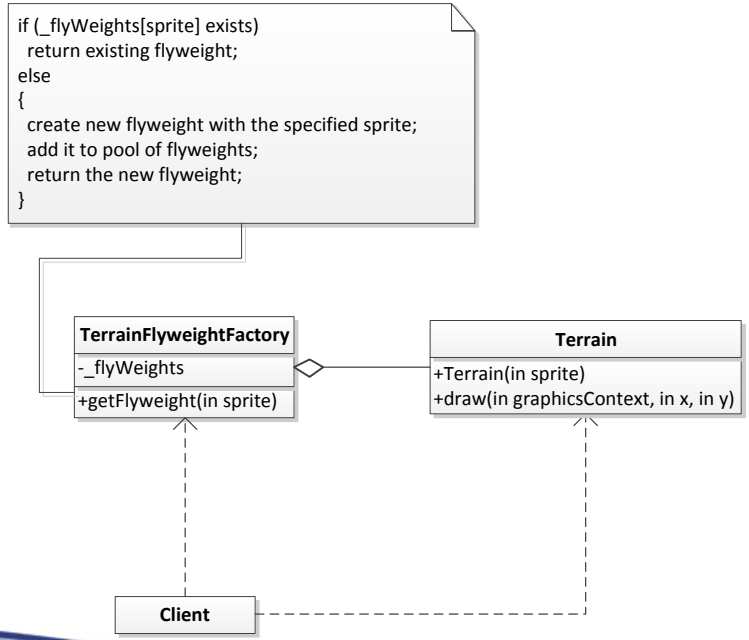
- With Flyweight:

```
GraphicsContext* gc = getGraphicsContext();
Terrain* t = flyweightFact->getFlyweight("tree", "tree.png");

for (int x = 0; x < 1000; ++x)
   for (int y = 0; y < 1000; ++y)
      t->draw(gc, x, y);
```

- 1 object created
- `tree.png` loaded and stored 1 time

Structural Patterns: Flyweight

- Further refinement:



Structural Patterns: Flyweight

```
GraphicsContext* gc = getGraphicsContext();
Terrain* t = flyweightFact->getFlyweight("tree.png");

for (int x = 0; x < 1000; ++x)
   for (int y = 0; y < 1000; ++y)
      t->draw(gc, x, y);

t = flyweightFact->getFlyweight("mountain.png");

for (int x = 1000; x < 1200; ++x)
   for (int y = 1000; y < 1200; ++y)
      t->draw(gc, x, y);
```

Structural Patterns: Flyweight

- Consequences:
  - The obvious: reduced memory requirements
  - May introduce run-time costs (transferring, finding, computing extrinsic state)
    - Should be offset by space savings
    - Savings increase as more flyweights are shared

Structural Design Patterns

- Adapter
- Bridge
- Composite
- Decorator
- Flyweight