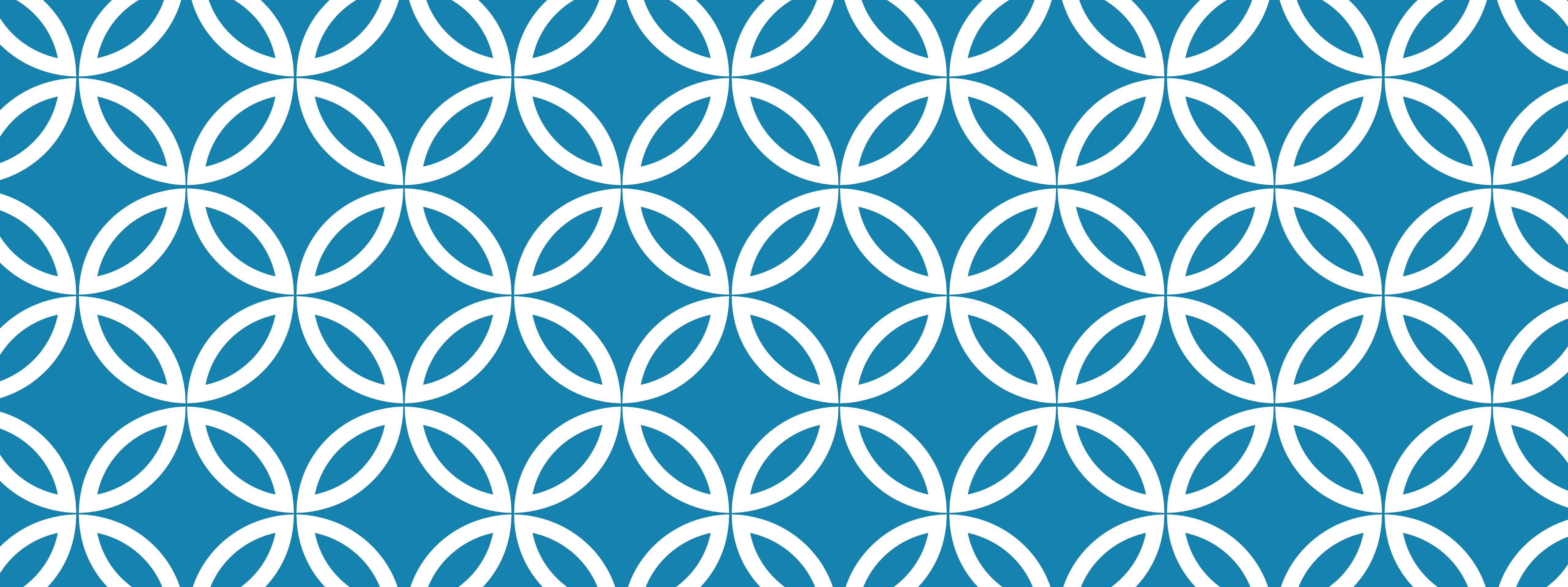


DESIGN PATTERNS – AN INTRODUCTION

Van den Driessche Willy

2015

AGENDA

- Introduction to design patterns
- An evolving example
- A note on efficiency
- Summary and questions

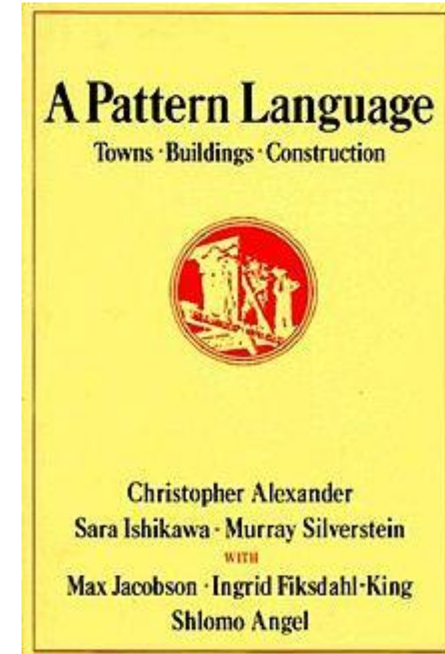
DESIGN PATTERNS BASICS

Christopher Alexander

A design pattern is the

1. proven
2. named
3. reusable form of
4. a solution to
5. a design problem.

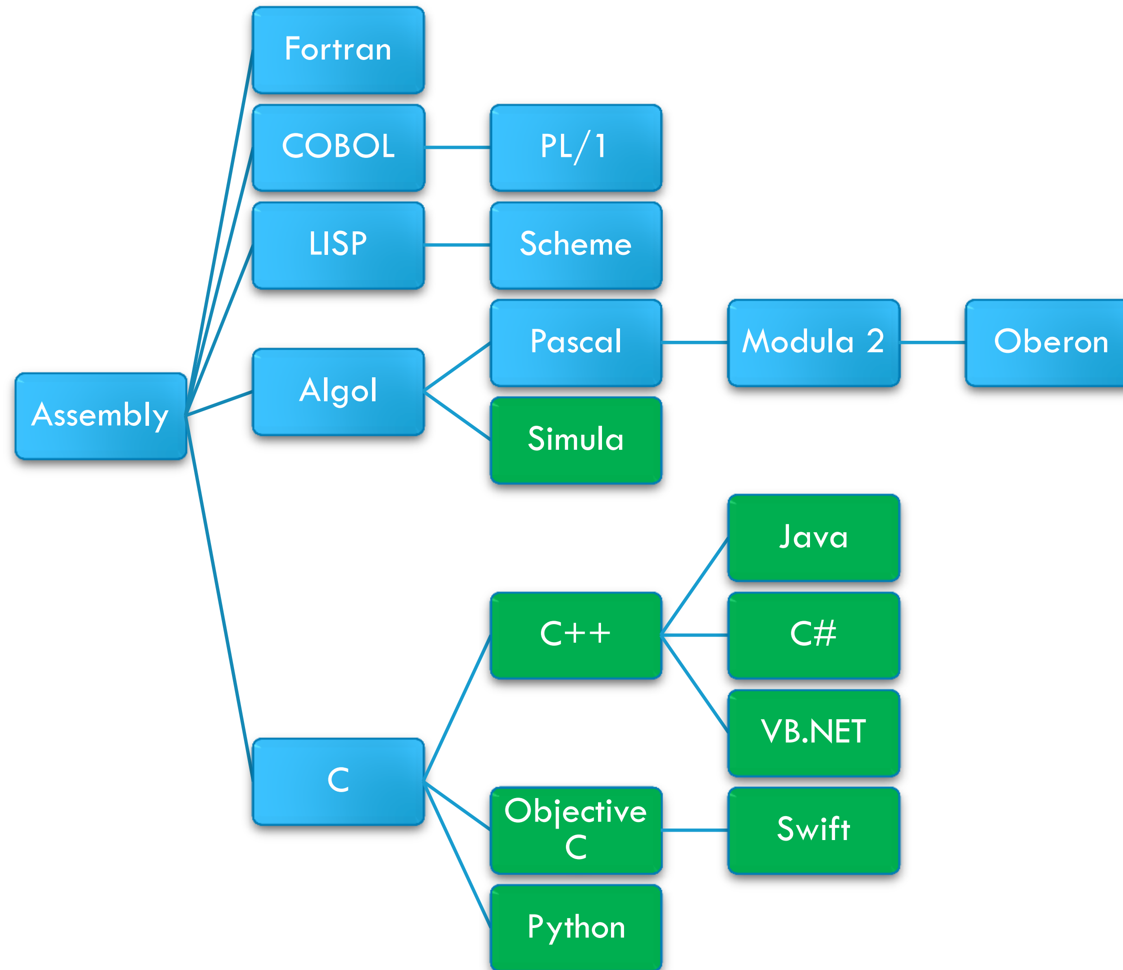
GoF book :



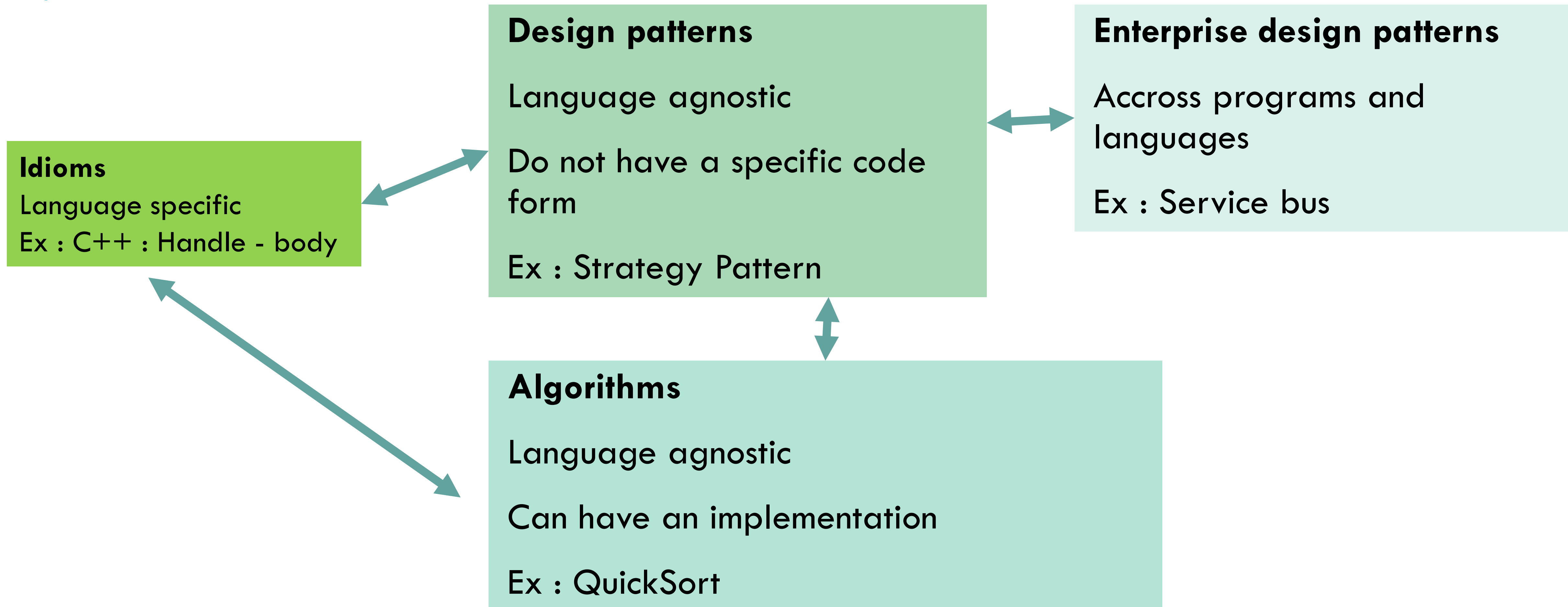
THE FORMAT : EXAMPLE

Name :	Strategy
Intent	Define a family of algorithms, encapsulate each one and make them interchangeable. Strategy lets the algorithm variate independent of the clients that use them.
Example	
Applicability	
Consequences	
Structure	

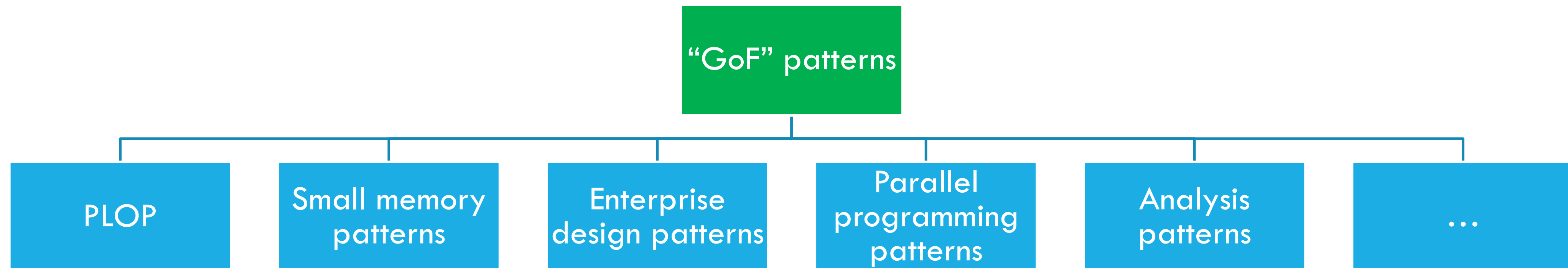
PATTERNS : OO LANGUAGES



PATTERNS AMONG THE BAG OF TRICKS



KINDS OF PATTERNS



EXAMPLE : TSP

```
problem* prob = new problem(std::string("city.txt"));
tour* t = new tour(prob);
…
while (((clock() - time)) < 10 * CLOCKS_PER_SEC)
{
tour* localImprovementSolution = t;
inspectNeighbourhood(workingSolution, localImprovementSolution);
if (tourcompare(localImprovementSolution, bestSolution) == 1)
▪ bestSolution->set(localImprovementSolution);
if (tourAccepted(localImprovementSolution, workingSolution))
▪ workingSolution->set(localImprovementSolution);
else
▪ localImprovementSolution->set(workingSolution);
}
```

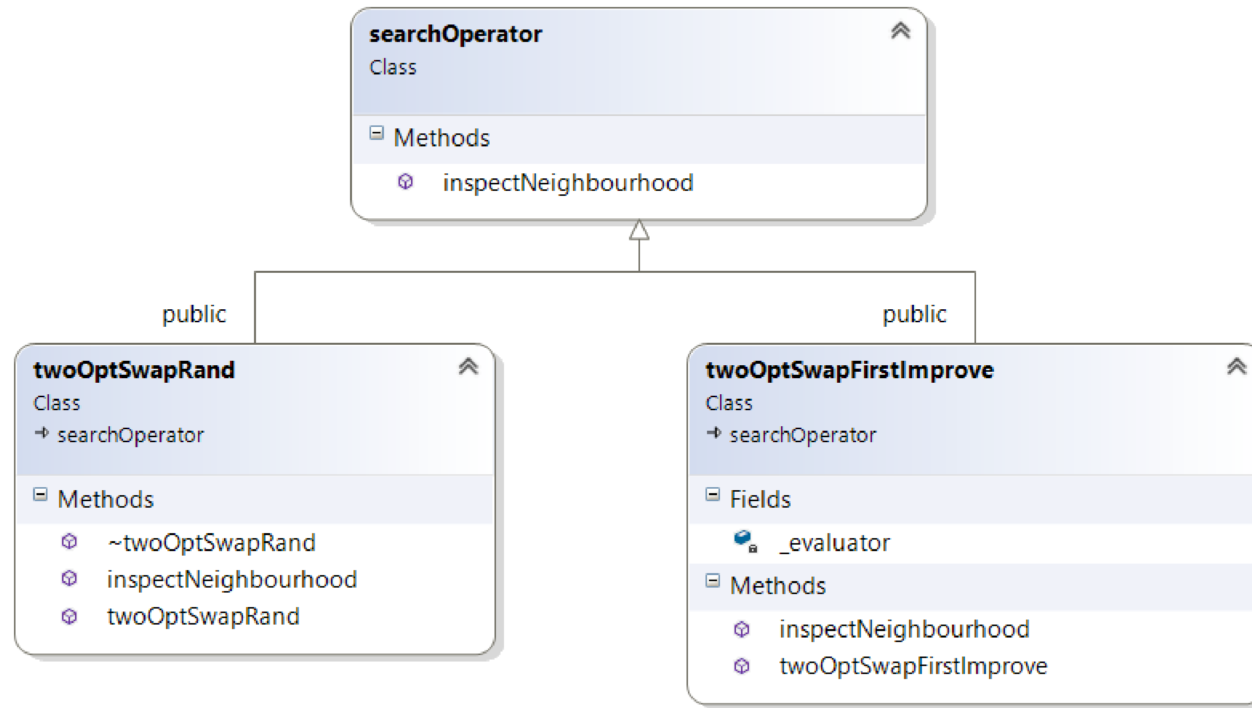
FICTIVE PROBLEM 1 : WE WANT TO EXPERIMENT WITH DIFFERENT IMPLEMENTATIONS

```
problem* prob = new problem(std::string("city.txt"));
tour* t = new tour(prob);
…
while (((clock() - time)) < 10 * CLOCKS_PER_SEC)
{
tour* localImprovementSolution = t;
inspectNeighbourhood(workingSolution, localImprovementSolution);
if (tourcompare(localImprovementSolution, bestSolution) == 1)
  ▪ bestSolution->set(localImprovementSolution);
if (tourAccepted(localImprovementSolution, workingSolution))
  ▪ workingSolution->set(localImprovementSolution);
else
  ▪ localImprovementSolution->set(workingSolution);
}
```

USING THE STRATEGY PATTERN : A SEARCH OPERATOR

```
class searchOperator{
public:
virtual void inspectNeighbourhood(
    tour* currentSolution, tour *neighbour) = 0;
};
```

SEARCH OPERATOR



TWOOPTSWAPRAND

```
 void twoOptSwapRand::inspectNeighbourhood( tour* currentTour, tour* neighbour){

int randomCitySource = rand()%(neighbour->getInstanceSize());
int randomCityDest = rand()%(neighbour->getInstanceSize());

 while( (randomCityDest = rand()%(neighbour->getInstanceSize())) ==
randomCitySource);

 if(randomCityDest < randomCitySource){swap (&randdomCityDest, randomCitySource)};

  while(randomCityDest>randomCitySource){

 neighbour->swapCities(randomCitySource++,randomCityDest--);

 }

 }
```

TWOOPTSWAPFIRSTIMPROVE

```
void twoOptSwapFirstImprove::inspectNeighbourhood(tour* currentTour, tour* neighbour){
    int instanceSize = currentTour->getInstanceSize();
    int idxSrc, idxDest;
    for(int sourceCity = 0; sourceCity < instanceSize-1; sourceCity++) {
         for(int destCity = sourceCity+1; destCity < instanceSize; destCity++) {
            idxSrc = sourceCity;  idxDest = destCity;
            while(idxDest > idxSrc){
                neighbour->swapCities(idxSrc++, idxDest--);
            }
           if(_evaluator->compare(neighbour, currentTour) > -1){ // improve or equal tours are allowed.
              return;} else {
                neighbour->set(currentTour);
            }
        }
    }
```

IMPORTANT DETAIL

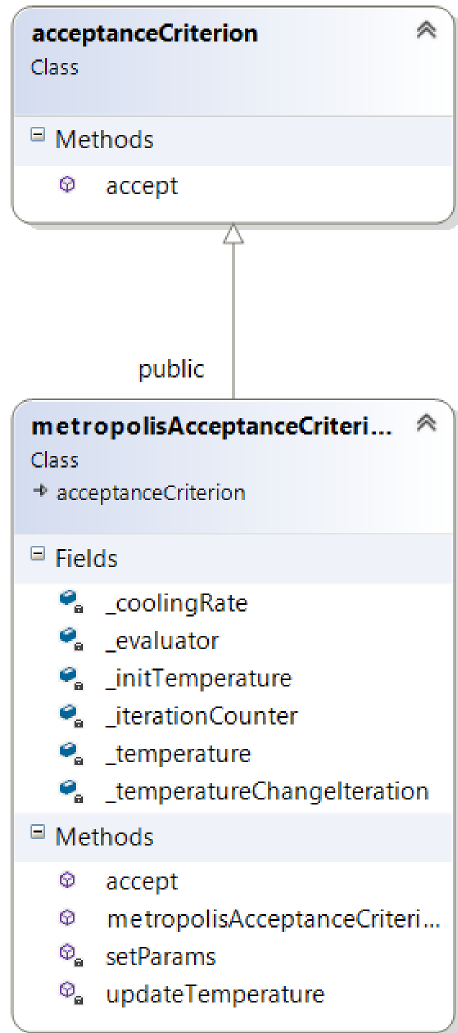
twoOptSwapRand and twoOptSwapFirstImprove both implement the same "interface". However, they are initialized in a different way.

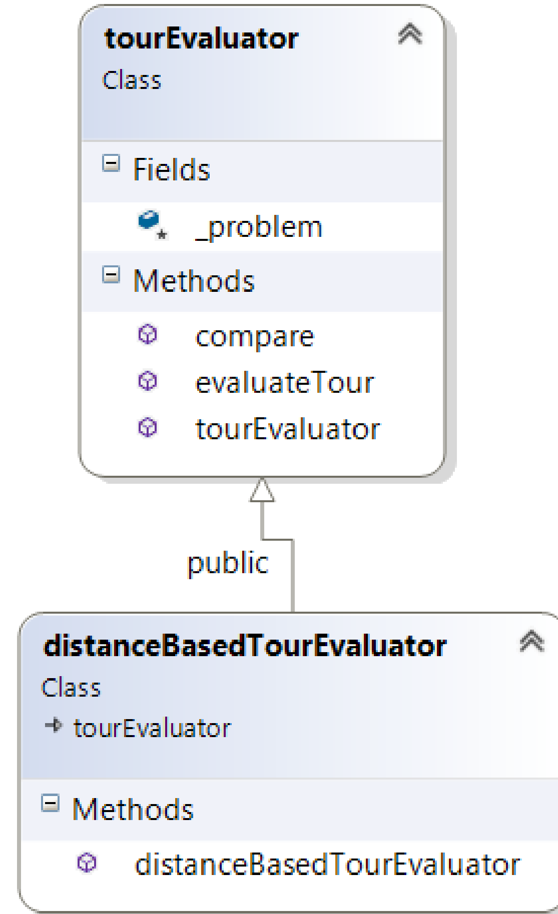
twoOptSwapFirstImprove::twoOptSwapFirstImprove(tourEvaluator* te)
{
_evaluator = te;
}
	twoOptSwapRand::twoOptSwapRand(){}

No problem : the constructor does not need to be the same.

Avoid adding parameters that are only needed for one child

OTHER STRATEGIES





WHAT HAVE WE WON ?

```
void searchAlgorithm::localSearch(double timeSeconds){
    double time = clock();
     while((clock()-time)/CLOCKS_PER_SEC < timeSeconds){
        searchop->inspectNeighbourhood(_workingSolution, _localImprovementSolution);
        if(evaluator->compare(_localImprovementSolution, _bestSolution) == 1){
            _bestSolution->set(_localImprovementSolution);
        }
        if(acceptance->accept(_localImprovementSolution, _workingSolution)){
           _workingSolution->set(_localImprovementSolution);
       }else {
           _localImprovementSolution->set(_workingSolution);
       }
    }
```

- There is a risk that we actually recognize the algorithm
- We can experiment with other search operator, other evaluations, … without changing a single line of code
- Individual strategy implementations are usually small and can be easily understood

CALLING LOCAL SEARCH

```
tour *t = new tour(prob);

tourEvaluator *te = new distanceBasedTourEvaluator(prob);
acceptanceCriterion *mac =
      new metropolisAcceptanceCriterion(te, prob);

searchOperator *so = new twoOptSwapRand();

localSearch * search = new localSearch(so, te, mac);

tour *best = search->LocalSearch(t,10);
```

FICTIVE PROBLEM 2 : WE WANT TO GATHER SOME STATISTICS

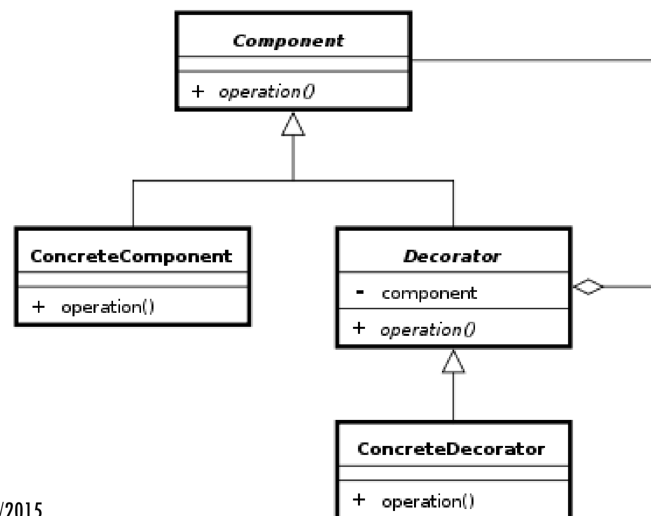
Suppose we want to know “how many times did we compare two tours” or

“how much time did we spend in the evaluation of tours”

Solution 1 : a decorator

DECORATOR

Name :	decorator
Intent	Attach additional responsabilities to an object dynamically. Decorators provide a flexible alternative to dubclassing for extending functionality



AN EXAMPLE DECORATOR

```
class countingSearchOperator : public searchOperator {
 public:
 countingSearchOperator(searchOperator * decoree)
 { _decoree = decoree;  _counter = 0; }

 void inspectNeighbourhood(tour* currentSolution, tour* neighbour) override {
 _counter++;   _decoree->inspectNeighbourhood(currentSolution, neighbour);
 }
 int getCounter() { return _counter;  }
 private:
 searchOperator* _decoree;  int _counter;
};
```

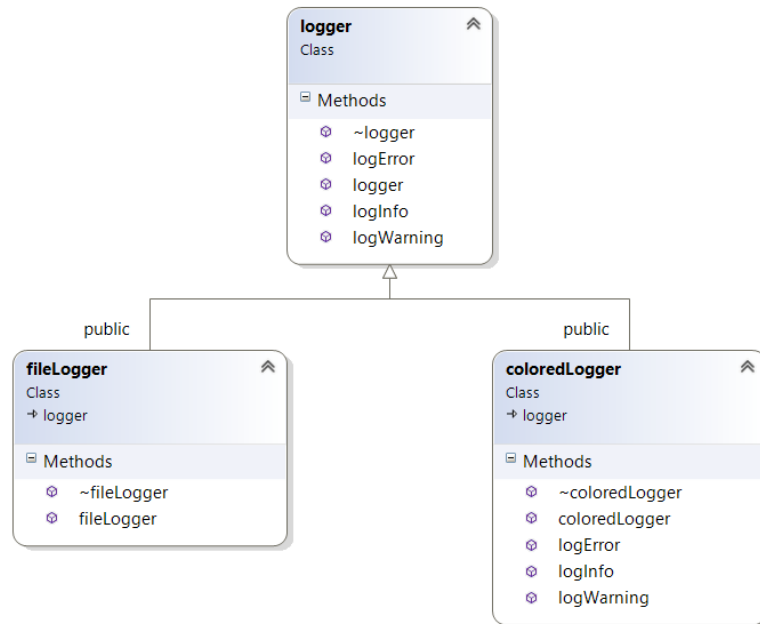
ADVANTAGES OF DECORATOR

- can add functionality without polluting original class.
- “pay per use”.   Only pay for overhead if needed.
- decorator code is usually small and readable.
- easy to wrap at runtime (ex with a command-line switch)

- use for “poor man’s aspect-oriented programming”

FICTIVE PROBLEM 3 : WE WANT TO ADD SOME LOGGING

- suppose we want to vary where we log to.



CHANGES TO LOCALSEARCH CLASS

```
localSearch::localSearch(searchOperator* searchOperator, tourEvaluator* tourEvaluator, acceptanceCriterion* acceptanceCriterion, logger* logger)
{
_searchOperator = searchOperator;
_tourEvaluator = tourEvaluator;
_acceptanceCriterion = acceptanceCriterion;
 _logger = logger;
}
```

CHANGES TO LOCAL SEARCH CLASS 2

```
if (NULL != _logger)
{
_logger->logInfo("Starting calculation");
}
```

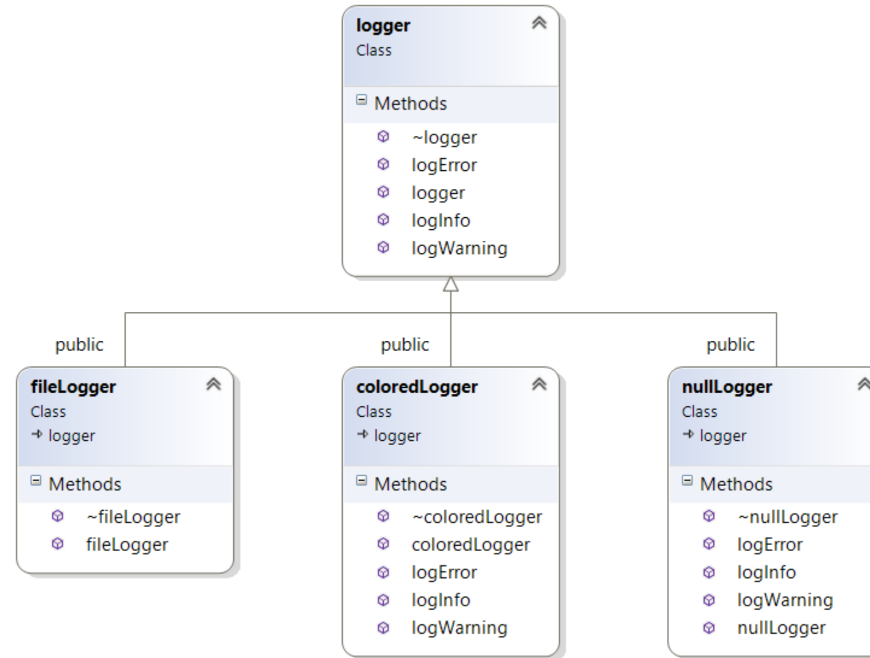
Problem : NULL checks pollute the code.

Solution : “Null Object” design pattern

NULL OBJECT DESIGN PATTERN (BOBBY WOOLF PLOP3)

Name	NullObject
Intent	The intent of a Null Object is to encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do nothing behavior. In short, a design where "nothing will come of nothing"

NULL LOGGER




```
class nullLogger :
public logger
{
public:
void logInfo(std::string message) override{}
void logWarning(std::string message) override {}
void logError(std::string message) override {}
nullLogger();
~nullLogger();
};
```

CHANGES TO LOCAL SEARCH CLASS 2

```
localSearch::localSearch(searchOperator* searchOperator,
tourEvaluator* tourEvaluator, acceptanceCriterion*
acceptanceCriterion, logger* logger)
{
_searchOperator = searchOperator;
_tourEvaluator = tourEvaluator;
_acceptanceCriterion = acceptanceCriterion;
  _logger = NULL != logger ? logger : new nullLogger();
}
```

NULL CHECKS ARE NOT NEEDED ANYMORE

```
_logger->logInfo("Starting calculation");
```

Code is cleaner

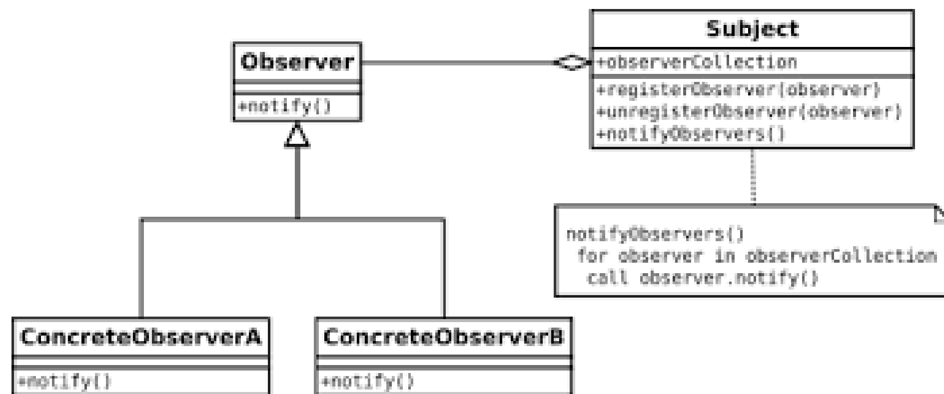
Performance penalty is acceptible.

FICTIVE PROBLEM 3 REVISITED : WE WANT TO ADD SOME LOGGING

There are alternative ways to introduce logging or other non-obstrusive operations

OBSERVER DESIGN PATTERN

Name	Observer
Intent	Define a one-to-many dependency between objects so that when one object changes state, all its dependencies are notified and updated automatically



```
_observer->Starting();…
while (((clock() - time)) < timeSeconds * CLOCKS_PER_SEC) {
…
if (_acceptanceCriterion->accept(localImprovementSolution, workingSolution))
{
_observer->SolutionAccepted(localImprovementSolution);
workingSolution->set(localImprovementSolution);
}
else
{
_observer->SolutionNotAccepted(workingSolution);
localImprovementSolution->set(workingSolution);
}
}…
_observer->Done(bestSolution);
```

A FICTITIOUS OBSERVER

```
class searchObserver
{
virtual void Starting()=0;
virtual void Done(tour* tour) = 0;
virtual void SolutionAccepted(tour* tour) = 0;
virtual void SolutionNotAccepted(tour* tour) = 0;
};
```

FICTIVE PROBLEM 4 : WE WANT TO COMBINE SEVERAL SEARCH OPERATORS

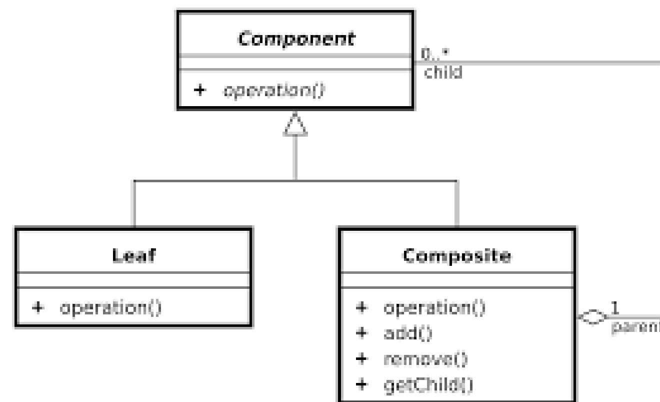
- we have several ways to search for search in the neigborhood.

We want to be able to combine them

Solution : Composite design pattern

COMPOSITE DESIGN PATTERN

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



AN EXAMPLE COMPOSITE

This implementation is not a PURE composite

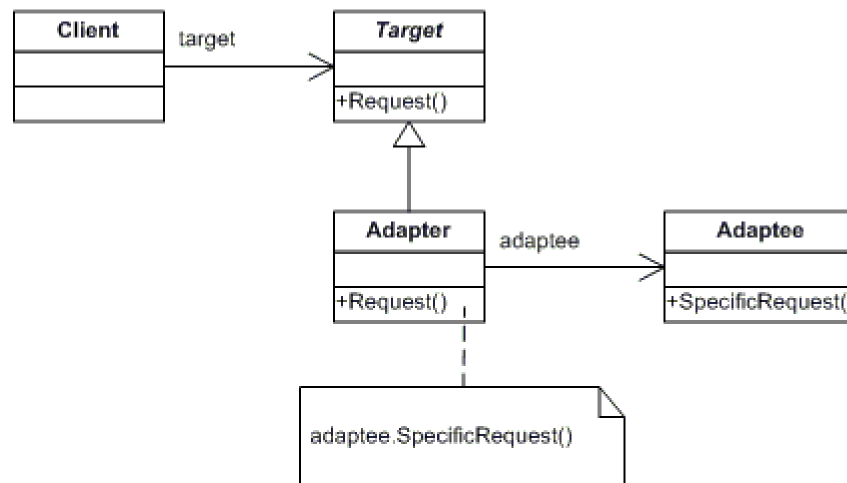
```
class compositeSearchOperator : public searchOperator
{
compositeSearchOperator(){  children = new std::vector<searchOperator*>(); }
void addChild(searchOperator* child){ _children->emplace_back(child); }
void inspectNeighbourhood(tour* currentSolution, tour* neighbour) override
  {
  int randomPosition = rand() % (_children->size());
  searchOperator* randomChild = _children->at(randomPosition);
  randomChild->inspectNeighbourhood(currentSolution, neighbour);
  }
};
```

FICTIVE PROBLEM 5 :
WE BOUGHT A LIBRARY WITH USEFUL ROUTINES

- we have bought a library of routines that work on TSP instances.
- They use a different representation

ADAPTER DESIGN PATTERN

Name	Adapter
Intent	Convert the interface of a class into another interface a client expects. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces



REAL PROBLEM 6 : WE ARE SO EXTENSIBLE THAT SIMPLE USAGE BECOMES PROBLEMATIC

We have created several strategy classes for searching, acceptance criteria and tour evalution.

We have created decorators for counting, logging, tracing …

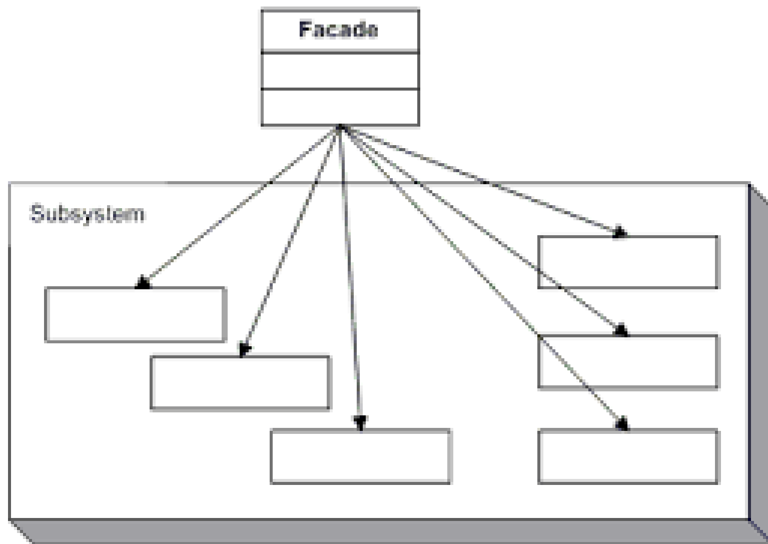
We have null objects.

We have adapters

Suppose a user just wants to solve a simple TSP : how can he use our classes without getting bogged down in details ?

FACADE DESIGN PATTERN

Name	Facade
Intent	Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.



BEFORE FACADE

```
tour *t = new tour(prob);

tourEvaluator *te = new countingTourEvaluator(new
distanceBasedTourEvaluator(prob));
metropolisAcceptanceCriterion *mac =new
loggingTourEvaluator(new
metropolisAcceptanceCriterion(te, prob));

searchOperator *so = new twoOptSwapRand();

localSearch * search = new localSearch(so, te, mac);

tour *best = search->LocalSearch(t,10);
```

AFTER FACADE

```
problem* prob = new problem(std::string("city.txt"));
tour *best = Tsp::SolveTsp(prob);
```

WHAT HAVE WE WON ?

- Facade makes simple usage simple.
- Facades are good for quick experiments, if you have reasonable defaults or if you simply don't care about the details.
- More sophisticated usage is still possible.
- Facade provides a lower "learning curve" for your subsystem.
- Overloading can reveal more options.

"simple things should be *simple*,
hard things should be *possible*"

PART 3 : A NOTE ON EFFICIENCY

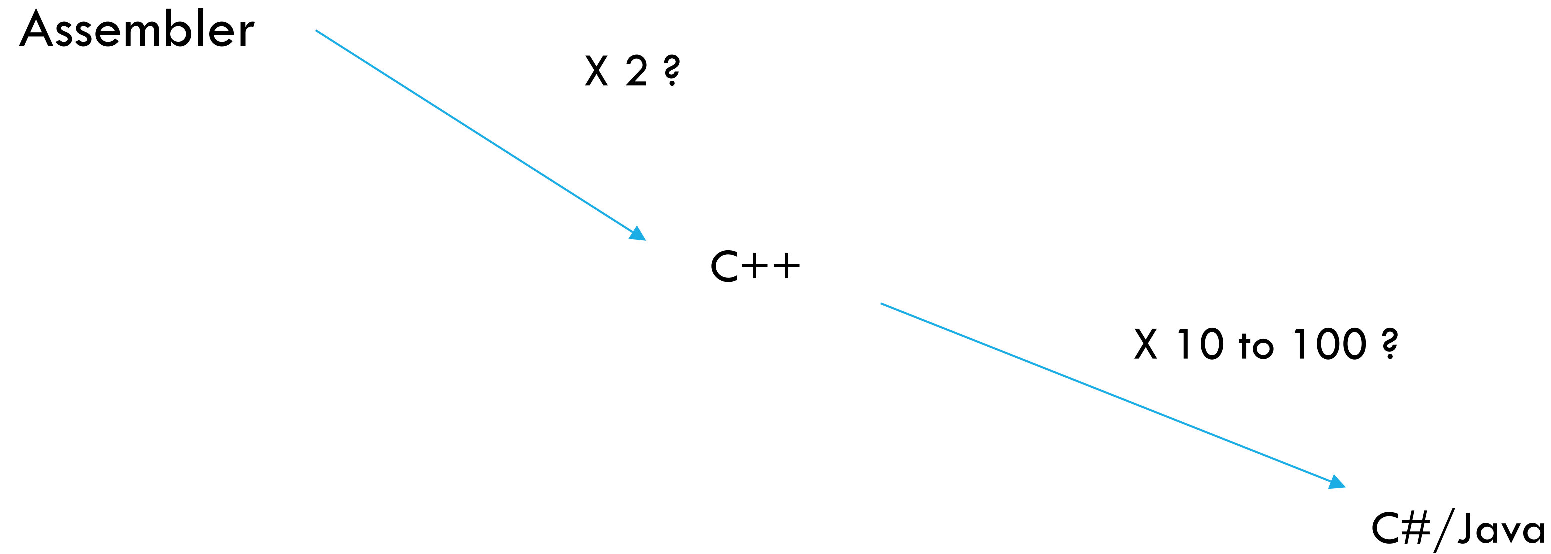
“Most of the time, the price of abstraction is performance.”

In general, design patterns only add a constant factor, they do not touch the “big O”

Object-oriented programming is an exceptionally bad idea which could only have originated in California.
Edsger Dijkstra

Certainly not every good program is object-oriented, and not every object-oriented program is good.
Bjarne Stroustrup

PROGRAMMING LANGUAGES AND EFFICIENCY



EFFICIENCT CODE IS LESS CHANGEABLE

QUESTIONS?