Class Design

Object-Oriented Programming with C++

Designing classes

- How to write classes in a way that they are easily understandable, maintainable and reusable

Class design: what to do?

- How many types of class do we need?
- When to define a class?
- What kind of interface/data in a class?
- Shall we construct inheritance to promote interface and code reuse?
- Which function should be virtual to support dynamic binding in run-time?

Contents

- Responsibility-driven design
- Coupling
- Cohesion
- Refactoring

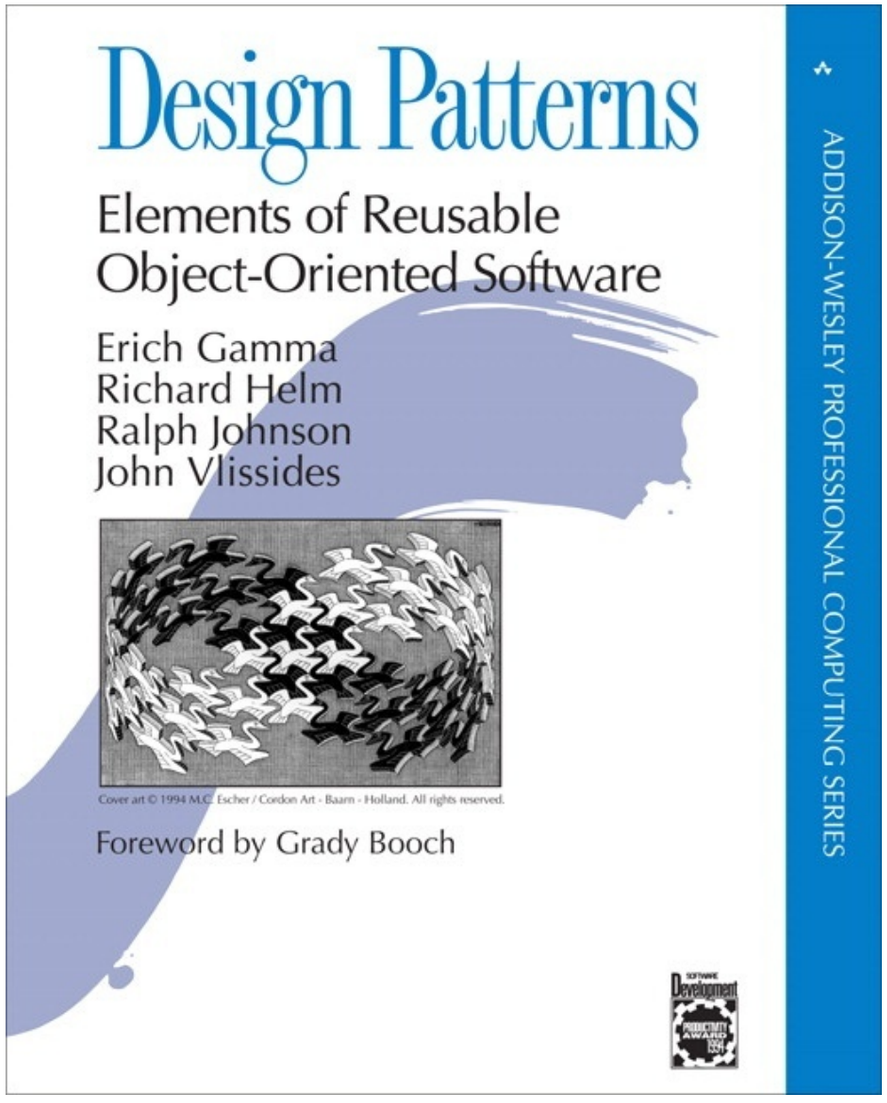
Software changes

- Software is not like a novel that is written once and then remains unchanged.
- Software is extended, corrected, maintained, ported, adapted…
- The work is done by different people over time (often decades).

Change or die

- There are only two options for software:
  - Either it is continuously maintained
  - Or it dies.
- Software that cannot be maintained will be thrown away.

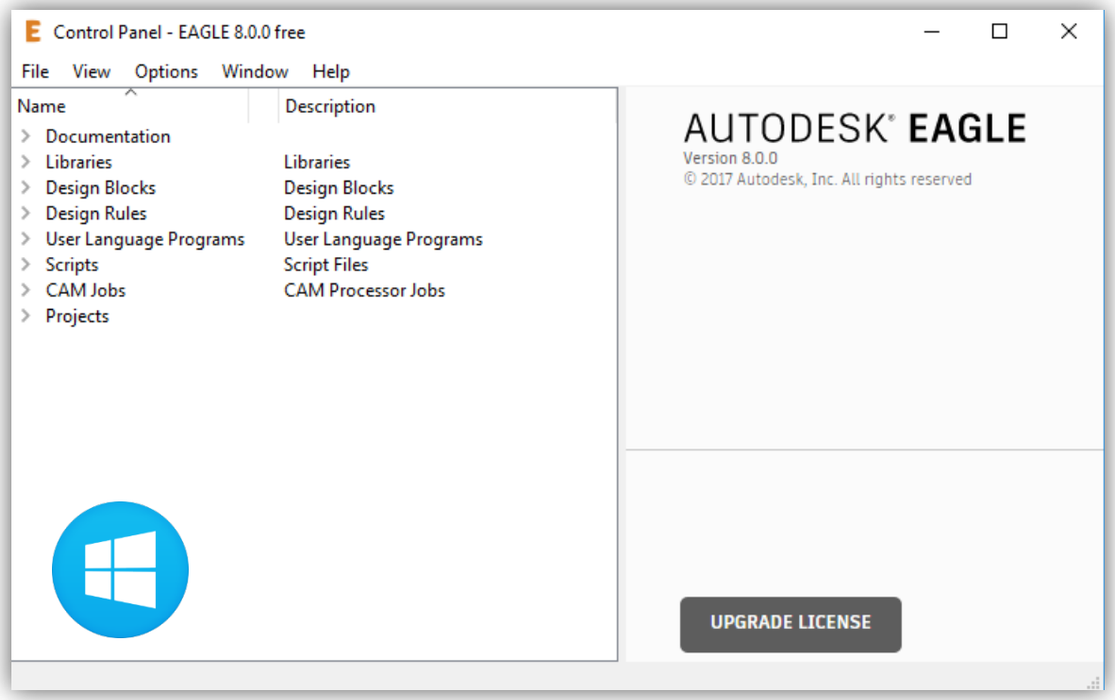
Design Patterns

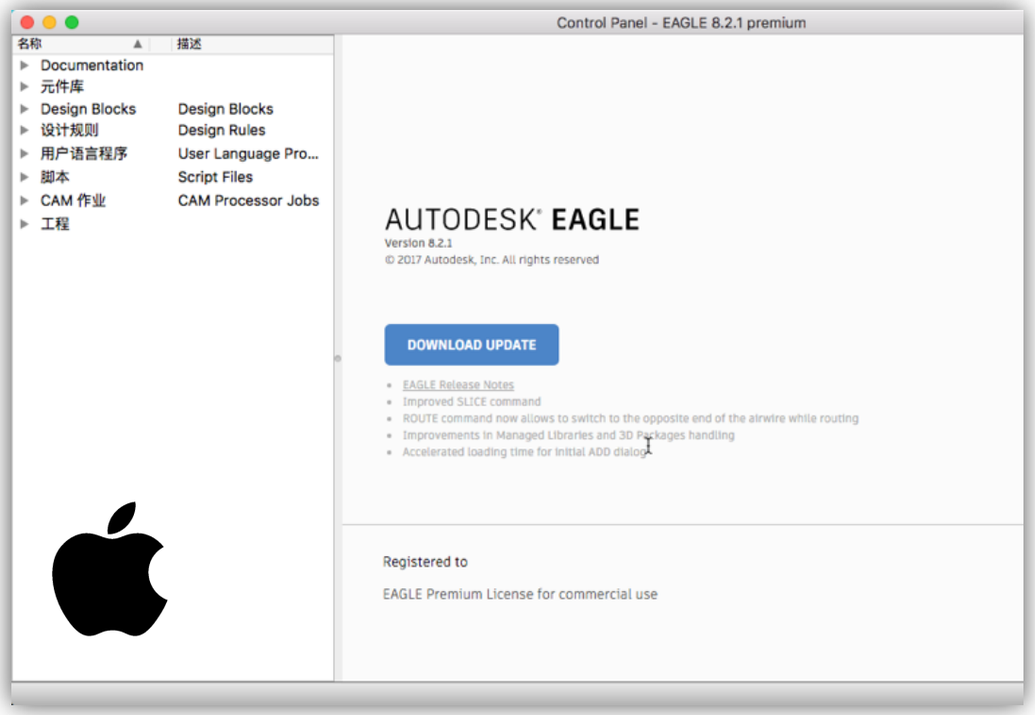


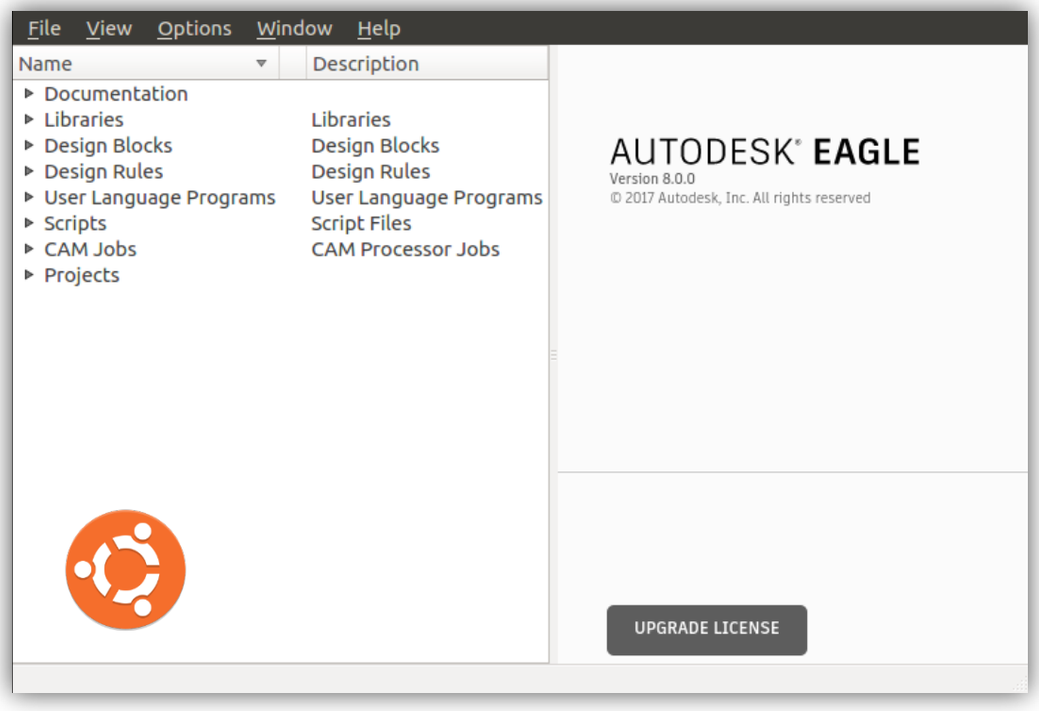
Design patterns

- Define a meaningful catalog:
  - 3 criterion, 23 design patterns
  - Creational, Structural, Behavioral
  - Aim for different purposes and contexts
- For each pattern:
  - name, classification
  - intent, motivation
  - structure, sample code
  - …

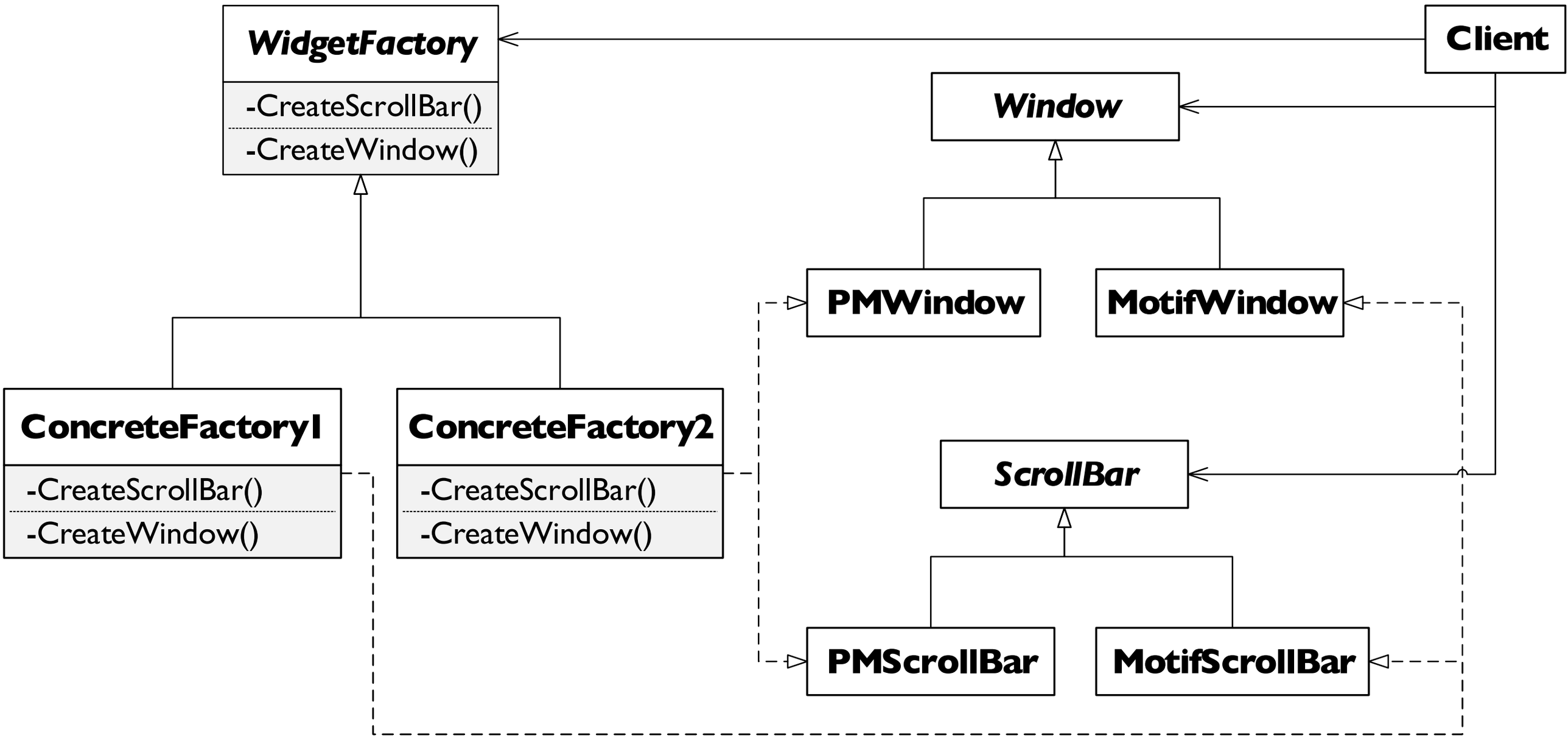
Multiple look-and-feel GUI



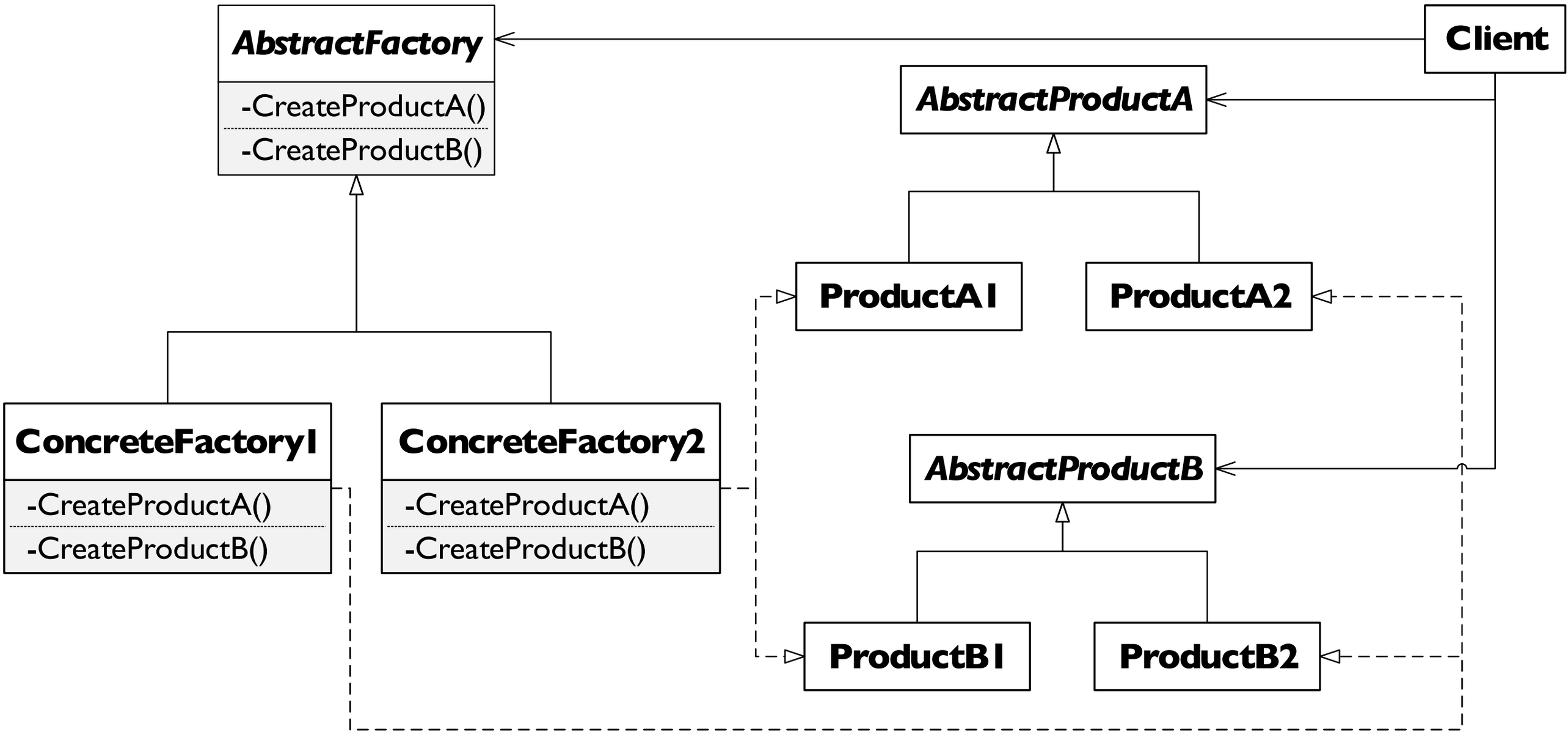




Multiple look-and-feel GUI



Abstract factory pattern

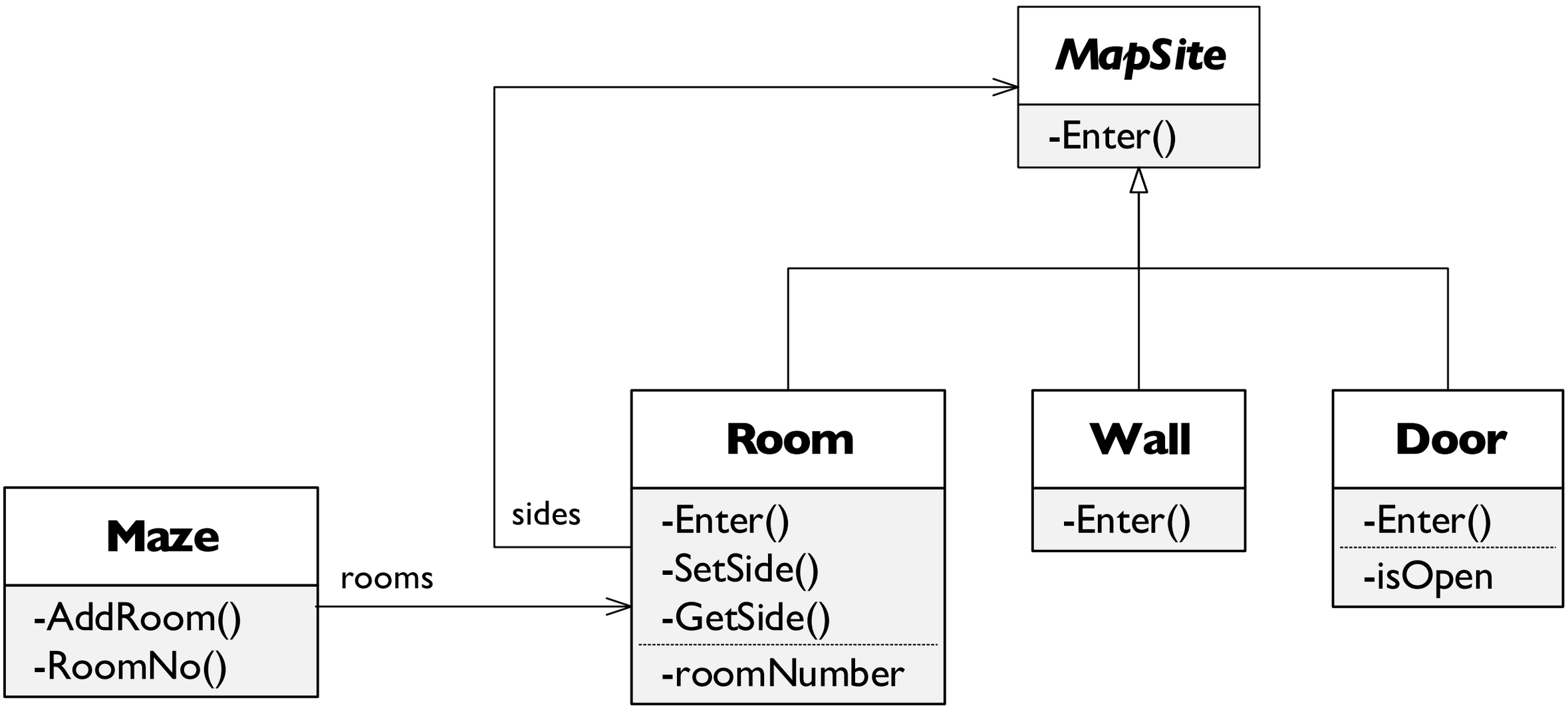


Building a maze game

- Sometimes the player is required to simply find the way out of a maze, given only a local view.
- Sometimes mazes contain problems to solve and dangers to overcome.
- Sometimes may provide a map of the part of the maze that has been explored.

Building a maze game

- Straightforward version: maze, room, wall, door



Building a maze game

```
class MapSite {
public:
   virtual void Enter() = 0; // pure virtual function
};
```

MapSite is an "*abstract*" type that cannot be instantiated, but can be used as a base class.

Building a maze game

```
class Room : public MapSite {
public:
   Room(int roomNo);
   MapSite* GetSide(Direction) const;
   void SetSide(Direction, MapSite*);

   virtual void Enter();
private:
   MapSite* _sides[4];
   int _roomNumber;
};
```

Building a maze game

```
class Wall : public MapSite {
public:
   Wall();
   virtual void Enter();
};
```

Building a maze game

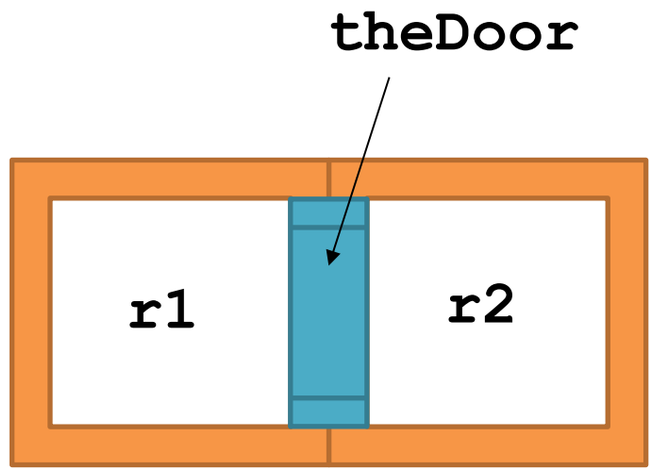
```
class Door : public MapSite {
public:
   Door(Room* = 0, Room* = 0);
   virtual void Enter();
   Room* OtherSideFrom(Room*);

private:
   Room* _room1;
   Room* _room2;
   bool _isOpen;
};
```

Building a maze game

```
class Maze {
public:
   Maze();
   void AddRoom(Room*);
   Room* RoomNo(int) const;

private:
   // …
};
```

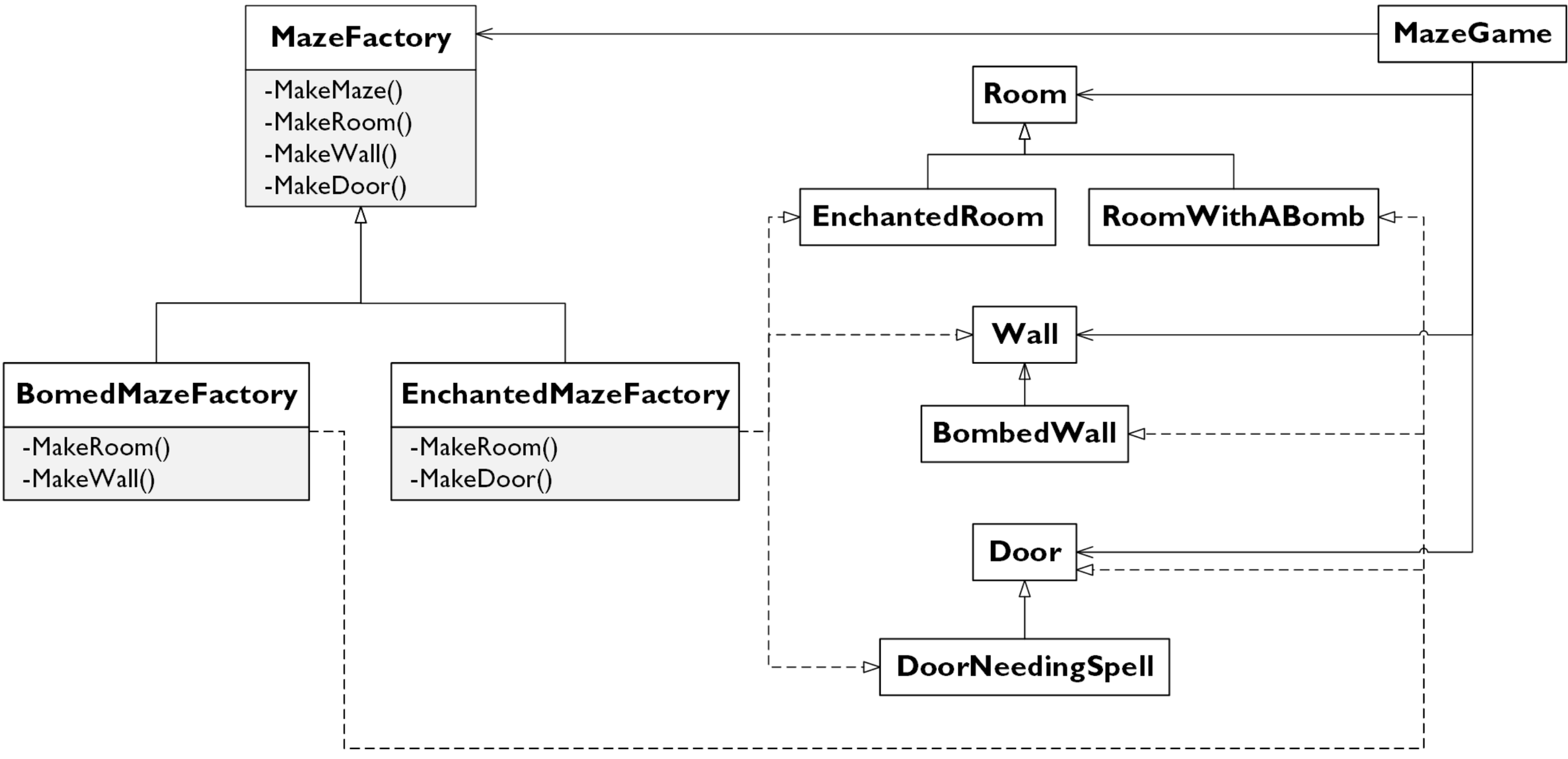
`RoomNo` does a look-up using a linear search, or a hash table …

```
Maze* MazeGame::CreateMaze() {
   Maze* aMaze = new Maze();
   Room* r1 = new Room(1);
   Room* r2 = new Room(2);
   Door* theDoor = new Door(r1, r2);

   r1->SetSide(East, theDoor);
   r2->SetSide(West, theDoor);
   // set other sides as walls
   r1->SetSide(South, new Wall);
   …

   aMaze->AddRoom(r1);
   aMaze->AddRoom(r2);
   return aMaze;
};
```



Hard-coded classes, inflexible!

Using abstract factory



Using abstract factory

```
class MazeFactory {
public:
  MazeFactory();
  virtual Maze* MakeMaze() const { return new Maze; }
  virtual Wall* MakeWall() const { return new Wall; }
  virtual Room* MakeRoom(int n) const
    { return new Room(n); }
  virtual Door* MakeDoor(Room* r1, Room* r2) const
    { return new Door(r1, r2); }
};
```

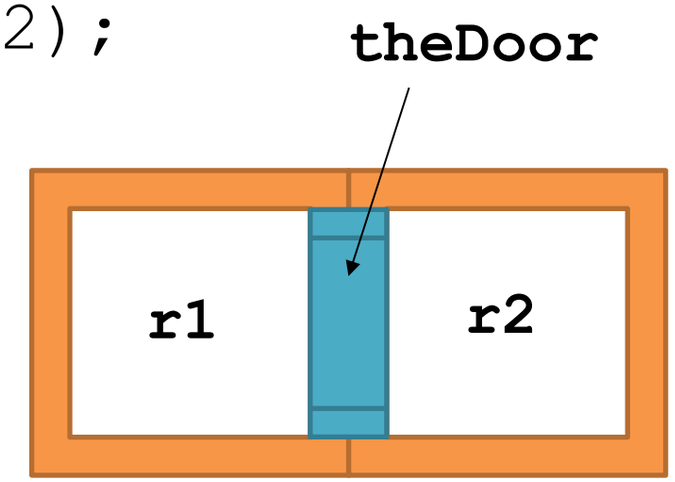
Using abstract factory

```
class EnchantedMazeFactory : public MazeFactory {
public:
  EnchantedMazeFactory();
  virtual Room* MakeRoom(int n) const
    { return new EnchantedRoom(n, CastSpell()); }
  virtual Door* MakeDoor(Room* r1, Room* r2) const
    { return new DoorNeedingSpell(r1, r2); }

Protected:
  Spell* castSpell() const;
};
```

Using abstract factory

```
class BombedMazeFactory : public MazeFactory {
public:
   BombedMazeFactory();

   virtual Door* MakeWall() const
      { return new BombedWall; }

   virtual Room* MakeRoom(int n) const
      { return new RoomWithABomb(n); }
};
```

```
Maze* MazeGame::CreateMaze(MazeFactory& factory) {
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* theDoor = factory.MakeDoor(r1, r2);

    r1->SetSide(East, theDoor);
    r2->SetSide(West, theDoor);
    // set other sides as walls
    r1->SetSide(South, factory.MakeWall());
    …

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);
    return aMaze;
};
```



Only depend on the input factory, flexible!

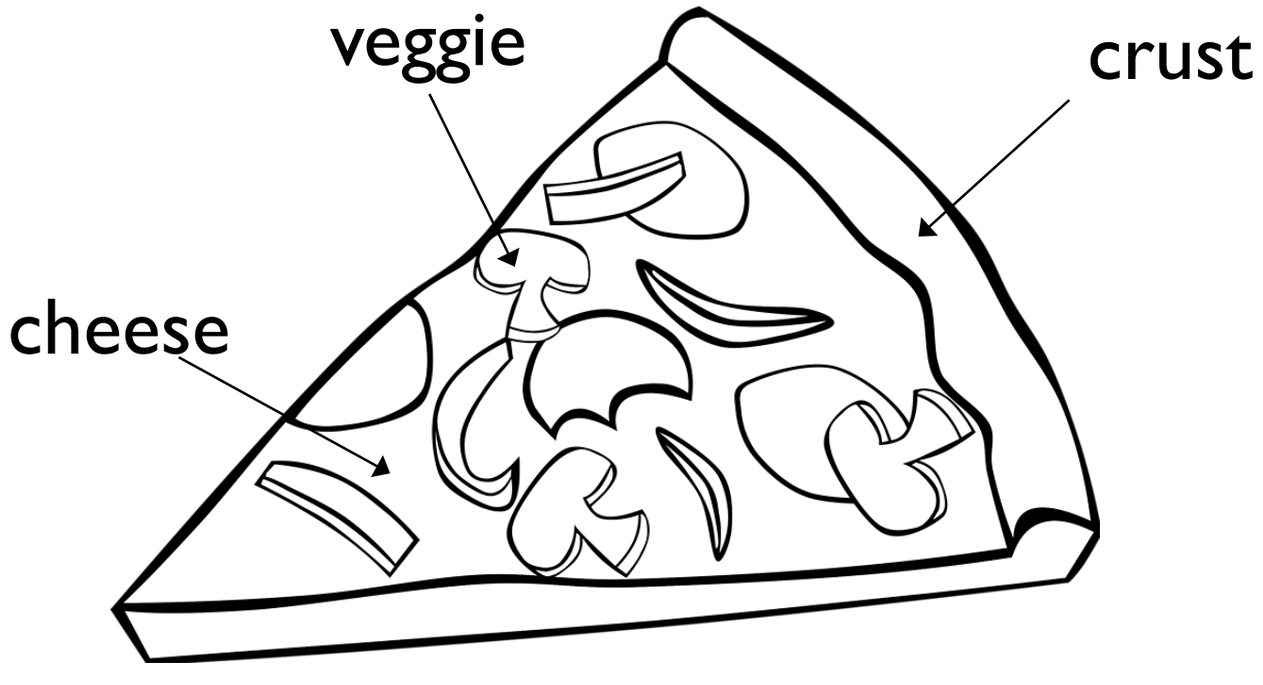
Using abstract factory

Now it's easy to create mazes with different components, e.g., build a maze that can contain *bombs*:

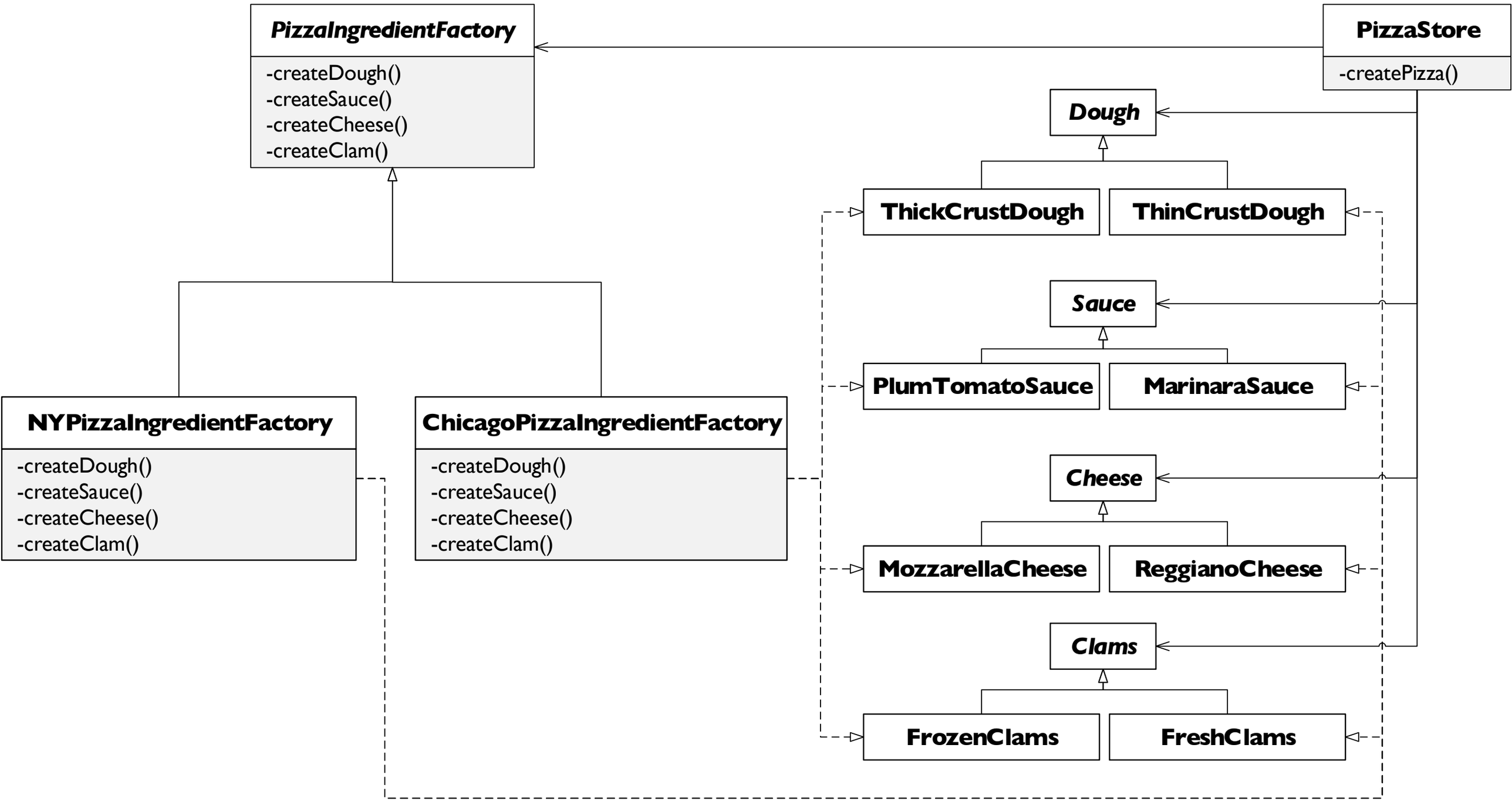
```
MazeGame game;
BombedMazeFactory factory;

Maze *m = game.CreateMaze(factory);
```

Open a pizza store

- Ingredients composition:
  - dough, sauce, cheese, veggie, pepperoni, clam, …
- Ingredients styles:
  - New York, Chicago, Rome, …



Open a pizza store



Open a pizza store

- Now taking an order of "cheese" pizza:

```
PizzaIngredientFactory factory =
    new NYPizzaIngredientFactory();
Pizza pizza = new CheesePizza(factory);
```

```
pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
```

Open for extension

Closed for modification

Design patterns

- Designing for change:
  - Anticipating new requirements and changes.
  - Identify what should be *variable*, and separate it.
  - Program to an interface, not an implementation
  - Let a part of the design vary independently of others, thus making it more robust to a particular kind of change.

Code quality

- Two important concepts for quality of code:
  - Coupling
  - Cohesion

Coupling

- Coupling refers to links between separate units of a program.
  - If two classes depend closely on many details of each other, we say they are *tightly coupled*.
  - We aim for *loose coupling*.

If X changes → how much code in Y must be changed?

Loose coupling

- Loose coupling makes it possible to:
  - Understand one class without reading others;
  - Change one class without affecting others;
  - Thus improves maintainability.

Tech. to loose

- call-back
- message mech.
- interface abstraction
- …

Cohesion

- Cohesion refers to the number and diversity of tasks that a single unit is responsible for.
- If each unit is responsible for one single logical task, we say it has high cohesion.
- Cohesion applies to classes and methods.
- We aim for high cohesion.

High cohesion

- High cohesion makes it easier to:
  - Understand what a class or method does;
  - Use descriptive names;
  - Reuse classes or methods.

Cohesion of methods/classes

- A method should be responsible for one and only one well defined task.
- A class should represent one single, well defined entity.

Code duplication

- Code duplication:
  - is an indicator of bad design;
  - makes maintenance harder;
  - leads to severe errors during maintenance.

Thinking ahead

- When designing a class, we try to think what changes are likely to be made in the future.
- We aim to make those changes easy.

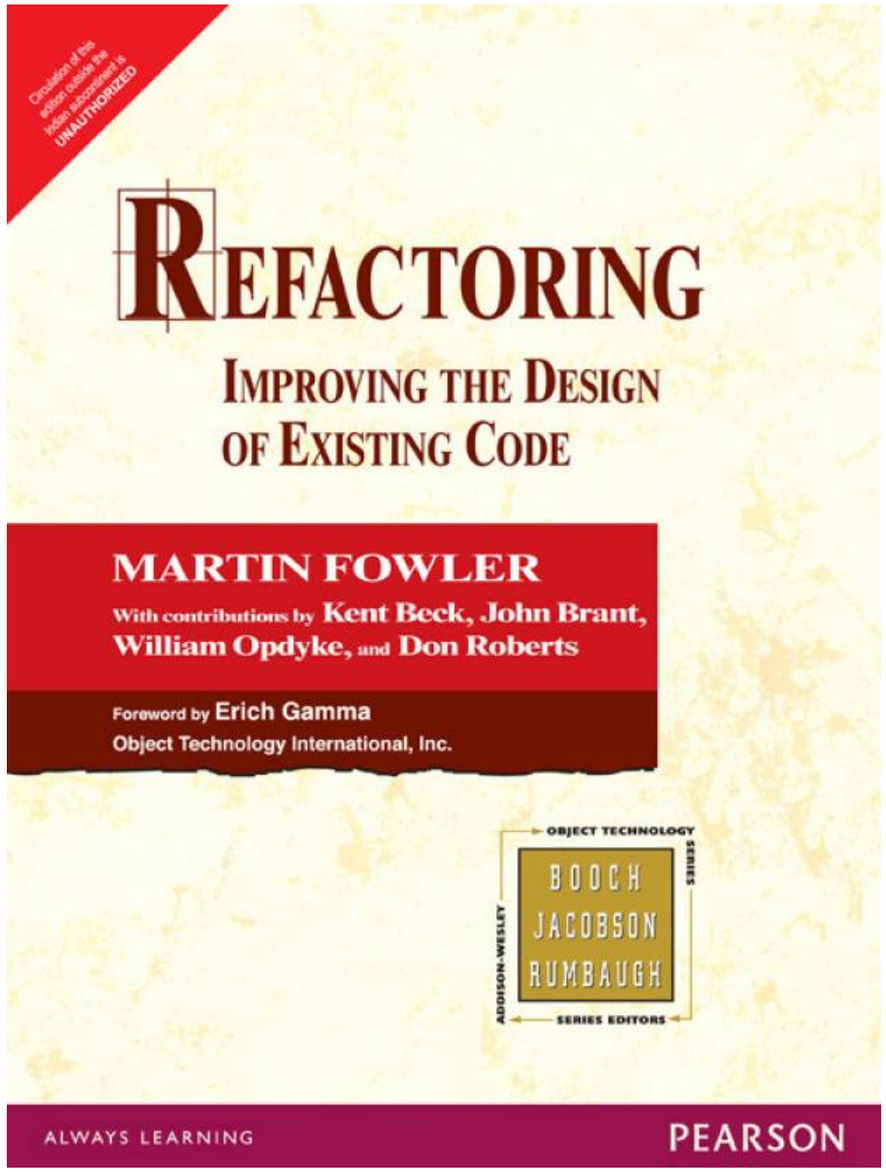
Refactoring

- When classes are maintained, often code is added.
- Classes and methods tend to become longer.
- Every now and then, classes and methods should be *refactored* to maintain cohesion and low coupling.

Refactoring and testing

- When refactoring code, separate the refactoring from making other changes.
- First do the refactoring only, without changing the functionality.
- Test before and after refactoring to ensure that nothing was broken.

Refactoring



Motivation

- Publisher + Subscribers
  - RSS, email
- Weather station data + Display device
  - Data: temperature, humidity, pressure
  - Display: current conditions, statistics, forecast
- Database + Spreadsheet/Chart
  - table, bar chart, pie chart

Motivation

```
class CurrentConditionDisplay {
public:
  void updateCurrentConditionData() {
    while (true) {
      float temp = weatherData.getTemperature();
      if (_temperature != temp) {
        _temperature = temp;
        // update displays …
      }
      // do the same for humidity and pressure
    }
};
```

Simple update from subscribers.

Motivation

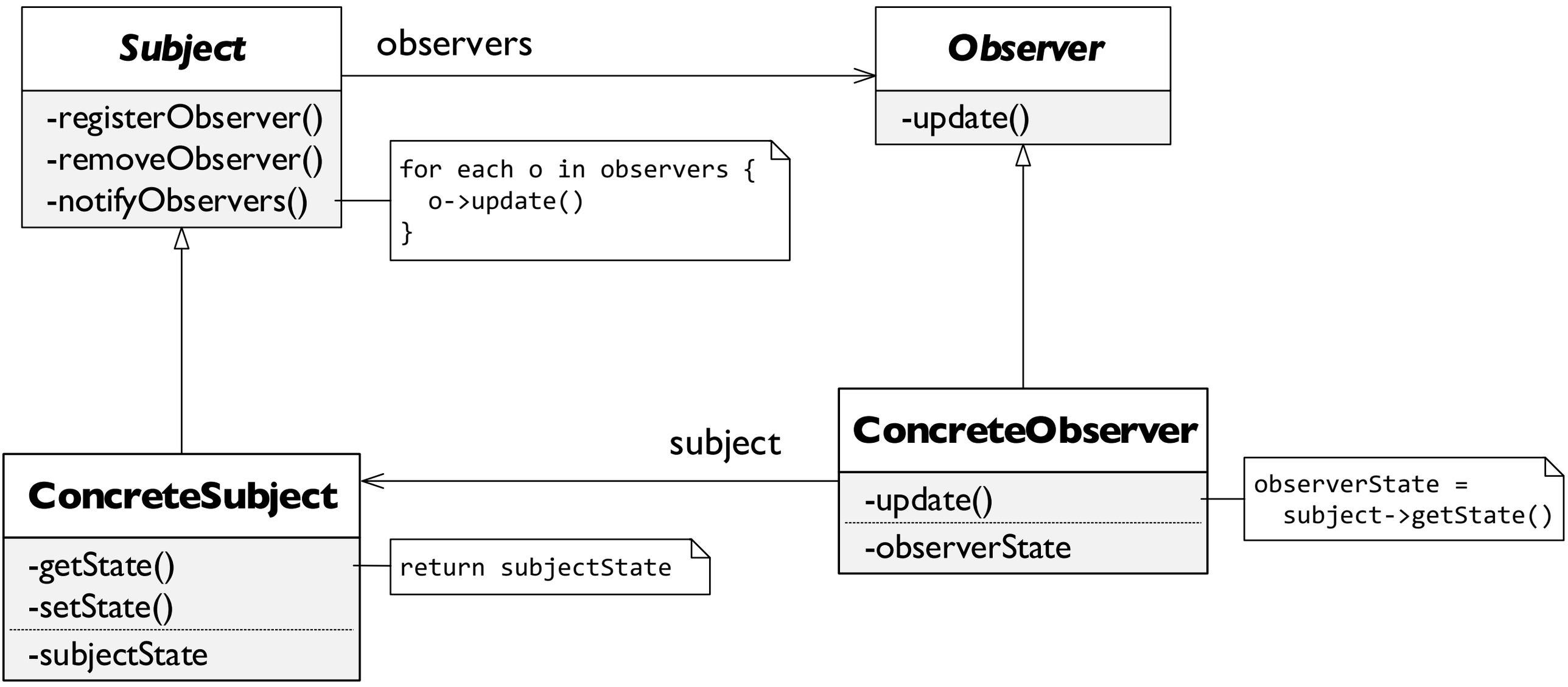
```
class WeatherData {
public:
  void measurementsChanged() {
    float temp = getTemperature();
    float humidity = getHumidity();
    float pressure = getPressure();

    currentConditionDisplay.update(temp, humidity, pressure);
    statisticsDisplay.update(temp, humidity, pressure);
    forecastDisplay.update(temp, humidity, pressure);
};
```

Simple update from publisher.

Observer pattern

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

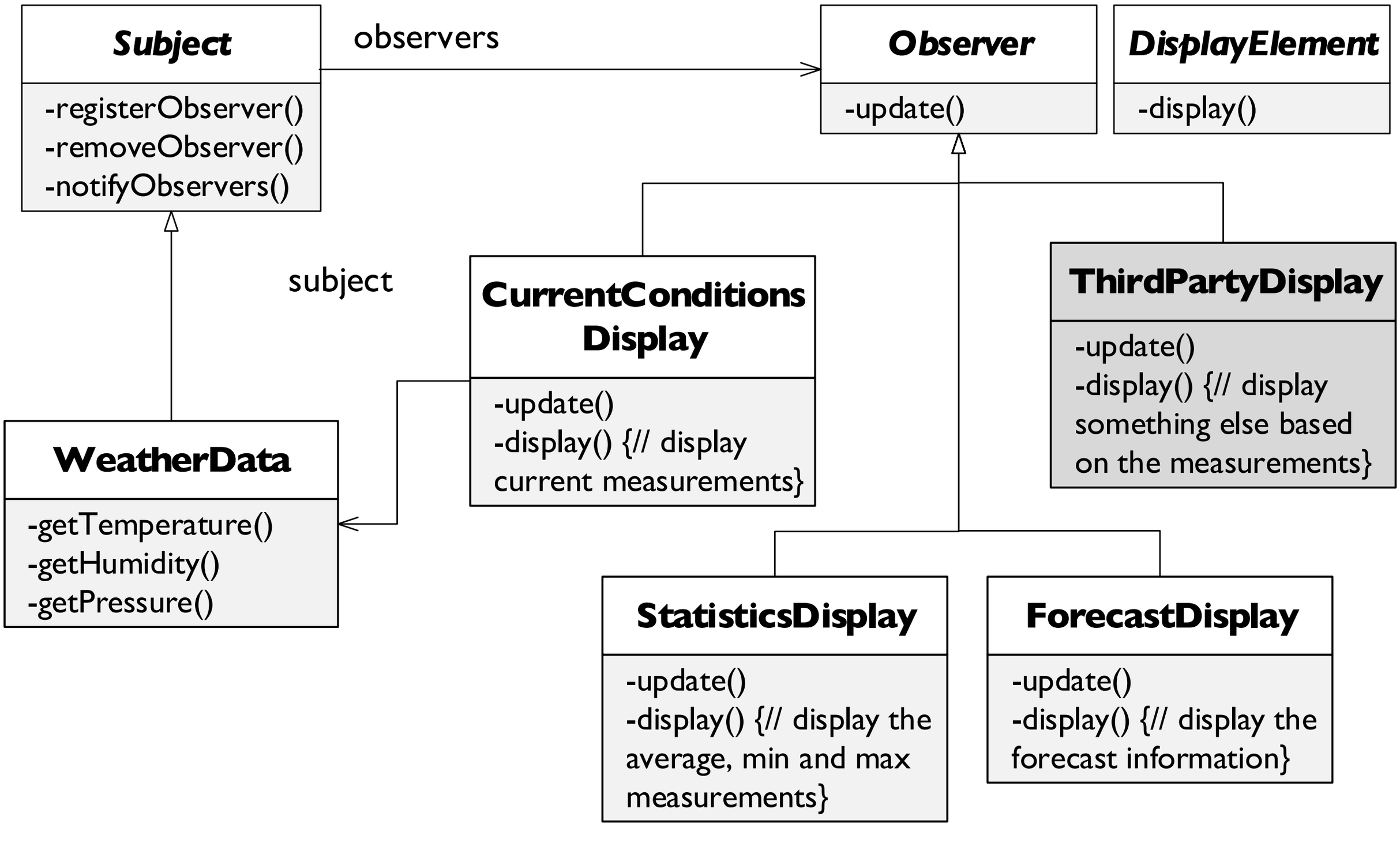
Observer pattern



Weather station design

- We have `WeatherData` as the *subject*
- We have many display devices as the *observers*:
  - `CurrentConditionsDisplay`
  - `StatisticsDisplay`
  - `ForecastDisplay`
  - `ThirdPartyDisplay`
  - …

Weather station design



Weather station design

```
class Subject {
public:
   virtual void registerObserver(Observer* o);
   virtual void removeObserver(Observer* o);
   virtual void notifyObservers();

private:
   std::vector<Observer*> _obs;
};
```

Weather station design

```
void Subject::registerObserver(Observer* o) {
  auto it = std::find(_obs.begin(), _obs.end(), o);
  if (it == _obs.end()) _obs.push_back(o);
}
void Subject::removeObserver(Observer* o) {
  auto it = std::find(_obs.begin(), _obs.end(), o);
  if (it != _obs.end()) _obs.erase(it);
}
void Subject::notifyObservers() {
  for (auto o : _obs)
    o->update();
}
```

Weather station design

```
class WeatherData : public Subject {
public:
   float getTemperature() { return _temperature; }
   float getHumidity() { return _humidity; }
   float getPressure() { return _pressure; }
   void setMeasurements() { …; notifyObservers(); }

private:
   float _temperature;
   float _humidity;
   float _pressure;
};
```

Weather station design

```
class Observer {
public:
   virtual ~Observer() {}
   virtual void update() = 0;
};
```

Weather station design

```
class CurrentConditionsDisplay
  : public Observer, public DisplayElement {
public:
  CurrentConditionsDisplay(WeatherData& wd);
  ~CurrentConditionsDisplay();
  void update() override;
  void display() override;

private:
  WeatherData& _wd; // ref to subject
  // … local states
};
```

Weather station design

```
CurrentConditionsDisplay(WeatherData& wd) : _wd(wd) {
   _wd.registerObserver(this);
}
~CurrentConditionsDisplay() {
   _wd.removeObserver(this);
}
```

Weather station design

```
void CurrentConditionsDisplay::update() {
  // save temp, humidity, pressure…
  _temp = _wd.getTemperature();
  _humidity = _wd.getHumidity();
  _pressure = _wd.getPressure();
  // display updated info on device
  display();
}

void CurrentConditionsDisplay::display() {
  // …
}
```

Weather station design

```
WeatherData wd;


CurrentConditionDisplay currentDisplay(wd);
StatisticsDisplay statisticsDisplay(wd);
ForecastDisplay forecastDisplay(wd);

wd.setMeasurements(); // modify the data, and every
                      // display device gets updated
```

```
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature: 80.0/82.0/78.0 degrees
Forecast: Watch out for cooler, rainy weather!
```

Clock timer design

```
class Timer : public Subject {
public:
   Timer();
   int getHour();
   int getMinute();
   int getSecond();

   void Tick() {
      // update internal time-keeping state …
      notifyObservers();
   }
};
```

Clock timer design

```
class DigitalClock
   : public Observer, public Widget {
public:
   DigitalClock(Timer& tm);
   ~DigitalClock();
   void update() override;
   void draw() override;

private:
   Timer& _tm; // ref to subject
   // … local states
};
```

Clock timer design

```
DigitalClock(Timer& tm) : _tm(tm) {
   _tm.registerObserver(this);
}

~DigitalClock() {
   _tm.removeObserver(this);
}
```

Clock timer design

```
void DigitalClock::update() {
   // update local states …
   _hr = _tm.getHour();
   _min = _tm.getMinute();
   _sec = _tm.getSecond();
   // draw the digital clock widget
   draw();
}

void DigitalClock::draw() {
   // …
}
```

Clock timer design

```
class AnalogClock
   : public Observer, public Widget {
public:
   AnalogClock(Timer& tm);
   ~AnalogClock();
   void update() override;
   void draw() override;
   // …
};
```

Clock timer design

```
Timer *tm = new Timer();


AnalogClock *analogClock = new AnalogClock(*tm);
DigitalClock *digitalClock = new DigitalClock(*tm);
```

```
tm->Tick(); // modify the time data, and every
            // clock widget gets re-drawed
```

Design questions

- Common questions:
  - How long should a class be?
  - How long should a method be?
- Can now be answered in terms of cohesion and coupling.

Design guidelines

- A method is too long if it does more then one logical task.
- A class is too complex if it represents more than one logical entity.
- Note: these are *guidelines* – they still leave much open to the designer.

SOLID principles

- In OO programming, the term ***SOLID*** represents five design principles intended to make software designs more ***understandable***, ***flexible*** and ***maintainable***.

SOLID principles

- **S**ingle responsibility principle
- **O**pen/closed principle

  software entities … should be open for extension, but closed for modification.

- **L**iskov substitution principle

  objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

- **I**nterface segregation principle
- **D**ependency inversion principle

  one should "depend upon abstractions, [not] concretions."

Review

- Programs are continuously changed.
- It is important to make this change possible.
- Quality of code requires much more than just performing correct at one time.
- Code must be readily understandable and maintainable.

Review

- Good quality code avoids duplication, displays high cohesion, low coupling.
- Coding style (commenting, naming, layout, etc.) is also important.
- There is a big difference in the amount of work required to change poorly structured and well structured code.