# Design Patterns

6.170 Lecture 18 Notes

Fall 2005

Reading: Chapter 15 of *Program Development in Java* by Barbara Liskov

## 1 Design patterns

A design pattern is:

- a standard solution to a common programming problem
- a technique for making code more flexible by making it meet certain criteria
- a design or implementation structure that achieves a particular purpose
- a high-level programming idiom
- shorthand for describing certain aspects of program organization
- connections among program components
- the shape of an object diagram or object model

### 1.1 Examples

Here are some examples of design patterns which you have already seen. For each design pattern, this list notes the problem it is trying to solve, the solution that the design pattern supplies, and any disadvantages associated with the design pattern. A software designer must trade off the advantages against the disadvantages when deciding whether to use a design pattern. Tradeoffs between flexibility and performance are common, as you will often discover in computer science (and other fields).

**Encapsulation (data hiding)**

**Problem:** Exposed fields can be directly manipulated from outside, leading to violations of the representation invariant or undesirable dependences that prevent changing the implementation.

**Solution:** Hide some components, permitting only stylized access to the object.

**Disadvantages:** The interface may not (efficiently) provide all desired operations. Indirection may reduce performance.

**Subclassing (inheritance)**

**Problem:** Similar abstractions have similar members (fields and methods). Repeating these is tedious, error-prone, and a maintenance headache.

**Solution:** Inherit default members from a superclass; select the correct implementation via run-time dispatching.

**Disadvantages:** Code for a class is spread out, potentially reducing understandability. Run-time dispatching introduces overhead.

**Iteration**

**Problem:** Clients that wish to access all members of a collection must perform a specialized traversal for each data structure. This introduces undesirable dependences and does not extend to other collections.

**Solution:** Implementations, which have knowledge of the representation, perform traversals and do bookkeeping. The results are communicated to clients via a standard interface.

**Disadvantages:** Iteration order is fixed by the implementation and not under the control of the client.

**Exceptions**

**Problem:** Errors occurring in one part of the code should often be handled elsewhere. Code should not be cluttered with error-handling code, nor return values preempted by error codes.

**Solution:** Introduce language structures for throwing and catching exceptions.

**Disadvantages:** Code may still be cluttered. It can be hard to know where an exception will be handled. Programmers may be tempted to use exceptions for normal control flow, which is confusing and usually inefficient.

These particular design patterns are so important that they are built into Java. Other design patterns are so important that they are built into other languages. Some design patterns may never be built into languages, but are still useful in their place.

## 1.2 When (not) to use design patterns

The first rule of design patterns is the same as the first rule of optimization: delay. Just as you shouldn't optimize prematurely, don't use design patterns prematurely. It may be best to first implement something and ensure that it works, then use the design pattern to improve weaknesses; this is especially true if you do not yet grasp all the details of the design. (If you fully understand the domain and problem, it may make sense to use design patterns from the start, just as it makes sense to use a more efficient rather than a less efficient algorithm from the very beginning in some applications.)

Design patterns may increase or decrease the understandability of a design or implementation. They can decrease understandability by adding indirection or increasing the amount of code. They can increase understandability by improving modularity, better separating concerns, and easing description. Once you learn the vocabulary of design patterns, you will be able to communicate more precisely and rapidly with other people who know the vocabulary. It's much better to say, "This is an instance of the visitor pattern" than "This is some code that traverses a structure and makes callbacks, and some certain methods must be present, and they are called in this particular way and in this particular order."

Most people use design patterns when they notice a problem with their design — something that ought to be easy isn't — or their implementation — such as performance. Examine the offending design or code. What are its problems, and what compromises does it make? What would you like to do that is presently too hard? Then, check a design pattern reference. Look for patterns that address the issues you are concerned with.

# 2 Creational patterns

## 2.1 Factories

Suppose you are writing a class to represent a bicycle race. A race consists of many bicycles (among other objects, perhaps).

```
class Race {

  Race createRace() {
    Frame frame1 = new Frame();
    Wheel frontWheel1 = new Wheel();
    Wheel rearWheel1 = new Wheel();
    Bicycle bike1 = new Bicycle(frame1, frontWheel1, rearWheel1);
    Frame frame2 = new Frame();
    Wheel frontWheel2 = new Wheel();
    Wheel rearWheel2 = new Wheel();
    Bicycle bike2 = new Bicycle(frame2, frontWheel2, rearWheel2);
    ...
  }

}
```

You can specialize `Race` for other bicycle races:

```
// French race
class TourDeFrance extends Race {

  Race createRace() {
    Frame frame1 = new RacingFrame();
    Wheel frontWheel1 = new Wheel700c();
    Wheel rearWheel1 = new Wheel700c();
    Bicycle bike1 = new Bicycle(frame1, frontWheel1, rearWheel1);
    Frame frame2 = new RacingFrame();
    Wheel frontWheel2 = new Wheel700c();
    Wheel rearWheel2 = new Wheel700c();
    Bicycle bike2 = new Bicycle(frame2, frontWheel2, rearWheel2);
    ...
  }

  ...
}

// all-terrain bicycle race
class Cyclocross extends Race {

  Race createRace() {
    Frame frame1 = new MountainFrame();
    Wheel frontWheel1 = new Wheel27in();
```

```
    Wheel rearWheel1 = new Wheel27in();
    Bicycle bike1 = new Bicycle(frame1, frontWheel1, rearWheel1);
    Frame frame2 = new MountainFrame();
    Wheel frontWheel2 = new Wheel27in();
    Wheel rearWheel2 = new Wheel27in();
    Bicycle bike2 = new Bicycle(frame2, frontWheel2, rearWheel2);
    ...
  }

  ...
}
```

In the subclasses, `createRace` returns a `Race` because the Java compiler enforces that overridden methods have identical return types.

For brevity, the code fragments above omit many other methods relating to bicycle races, some of which appear in each class and others of which appear only in certain classes.

The repeated code is tedious, and in particular, we weren't able to reuse method `Race.createRace` at all. (There is a separate issue of abstracting out the creation of a single bicycle to a function; we will use that without further discussion, as it is obvious, at least after 6.001.) There must be a better way. The Factory design patterns provide an answer.

### 2.1.1 Factory method

A factory method is a method that manufactures objects of a particular type.

We can add factory methods to `Race`:

```
class Race {

  Frame createFrame() { return new Frame(); }
  Wheel createWheel() { return new Wheel(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new Bicycle(frame, front, rear);
  }
  // return a complete bicycle without needing any arguments
  Bicycle completeBicycle() {
    Frame frame = createFrame();
    Wheel frontWheel = createWheel();
    Wheel rearWheel = createWheel();
    return createBicycle(frame, frontWheel, rearWheel);
  }

  Race createRace() {
    Bicycle bike1 = completeBicycle();
    Bicycle bike2 = completeBicycle();
    ...
  }

}
```

Now subclasses can reuse `createRace` and even `completeBicycle` without change:

```
// French race
class TourDeFrance extends Race {

  Frame createFrame() { return new RacingFrame(); }
  Wheel createWheel() { return new Wheel700c(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new RacingBicycle(frame, front, rear);
  }

}

class Cyclocross extends Race {

  Frame createFrame() { return new MountainFrame(); }
  Wheel createWheel() { return new Wheel26inch(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new RacingBicycle(frame, front, rear);
  }

}
```

The `create...` methods are called *factory methods*.

### 2.1.2 Factory object

If there are many objects to construct, including the factory methods in each class can bloat the code and make it hard to change. Sibling subclasses cannot easily share the same factory method.

A *factory object* is an object that encapsulates factory methods.

```
class BicycleFactory {
  Frame createFrame() { return new Frame(); }
  Wheel createWheel() { return new Wheel(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new Bicycle(frame, front, rear);
  }

  // return a complete bicycle without needing any arguments
  Bicycle completeBicycle() {
    Frame frame = createFrame();
    Wheel frontWheel = createWheel();
    Wheel rearWheel = createWheel();
    return createBicycle(frame, frontWheel, rearWheel);
  }
}

class RacingBicycleFactory {
  Frame createFrame() { return new RacingFrame(); }
```

```
  Wheel createWheel() { return new Wheel700c(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new RacingBicycle(frame, front, rear);
  }
}

class MountainBicycleFactory {
  Frame createFrame() { return new MountainFrame(); }
  Wheel createWheel() { return new Wheel26inch(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new RacingBicycle(frame, front, rear);
  }
}
```

The `Race` methods use the factory objects.

```
class Race {

  BicycleFactory bfactory;

  // constructor
  Race() {
    bfactory = new BicycleFactory();
  }

  Race createRace() {
    Bicycle bike1 = bfactory.completeBicycle();
    Bicycle bike2 = bfactory.completeBicycle();
    ...
  }

}

class TourDeFrance extends Race {
  // constructor
  TourDeFrance() {
    bfactory = new RacingBicycleFactory();
  }
}

class Cyclocross extends Race {
  // constructor
  Cyclocross() {
    bfactory = new MountainBicycleFactory();
  }
}
```

In this version of the code, the type of bicycle is still hard-coded into each variety of race. There is a more flexible method which requires a change to the way that clients call the constructor.

```
class Race {

  BicycleFactory bfactory;

  // constructor
  Race(BicycleFactory bfactory) {
    this.bfactory = bfactory;
  }

  Race createRace() {
    Bicycle bike1 = bfactory.completeBicycle();
    Bicycle bike2 = bfactory.completeBicycle();
    ...
  }

}

class TourDeFrance extends Race {
  // constructor
  TourDeFrance(BicycleFactory bfactory) {
    this.bfactory = bfactory;
  }
}

class Cyclocross extends Race {
  // constructor
  Cyclocross(BicycleFactory bfactory) {
    this.bfactory = bfactory;
  }
}
```

This is the most flexible mechanism of all. Now a client can control both the variety of race and the variety of bicycle used in the race, for instance via a call like

```
new TourDeFrance(new TricycleFactory())
```

One reason that factory methods are required is *the first weakness of Java constructors*: Java constructors always return an object of the specified type. They can never return an object of a subtype, even though that would be type-correct (both according to Java subtyping and according to true behavior subtyping as was described in Lecture 14).

In fact, `createRace` is itself a factory method.

### 2.1.3 Prototype

The prototype pattern provides another way to construct objects of arbitrary types. Rather than passing in a `BicycleFactory` object, a `Bicycle` object is passed in. Its `clone` method is invoked to create new bicycles; we are making copies of the given object.

```
class Bicycle {
```

```
  Object clone() { ... }
}

class Frame {
  Object clone() { ... }
}

class Wheel {
  Object clone() { ... }
}

class RacingBicycle {
  Object clone() { ... }
}

class RacingFrame {
  Object clone() { ... }
}

class Wheel700c {
  Object clone() { ... }
}

class MountainBicycle {
  Object clone() { ... }
}

class MountainFrame {
  Object clone() { ... }
}

class Wheel26inch {
  Object clone() { ... }
}

class Race {

  Bicycle bproto;

  // constructor
  Race(Bicycle bproto) {
    this.bproto = bproto;
  }

  Race createRace() {
    Bicycle bike1 = (Bicycle) bproto.clone();
    Bicycle bike2 = (Bicycle) bproto.clone();
    ...
```

```
  }

}

class TourDeFrance extends Race {
  // constructor
  TourDeFrance(Bicycle bproto) {
    this.bproto = bproto;
  }
}

class Cyclocross extends Race {
  // constructor
  Cyclocross(Bicycle bproto) {
    this.bproto = bproto;
  }
}
```

Effectively, each object is itself a factory specialized to making objects just like itself. Prototypes are frequently used in dynamically typed languages such as Smalltalk, less frequently used in statically typed languages such as C++ and Java.

There is no free lunch: the code to create objects of particular classes must go somewhere. Factory methods put the code in methods in the client; factory objects put the code in methods in a factory object; and prototypes put the code in `clone` methods.

# 3 Behavioral patterns

It is easy enough for a single client to use a single abstraction. (We have seen patterns for easing the task of changing the abstraction being used, which is a common task.) However, occasionally a client may need to use multiple abstractions; furthermore, the client may not know ahead of time how many or even which abstractions will be used. The observer, blackboard, and mediator patterns permit such communication.

## 3.1 Observer

Suppose that there is a database of all MIT student grades, and the 6.170 staff wishes to view the grades of 6.170 students. They could write a `SpreadsheetView` class that displays information from the database. (We will assume that the viewer caches information about 6.170 students — it needs this information in order to redraw, for example — but whether it does so is not an important part of this discussion.) The display might look something like this:

| | PS1 | PS2 | PS3 |
|---|---|---|---|
| G. Bates | 45 | 85 | 80 |
| A. Hacker | 95 | 90 | 85 |
| A. Turing | 90 | 100 | 95 |

Suppose the code to communicate between the grade database and the view of the database uses the following interface:

```
interface GradeDBViewer {
  void update(String course, String name, String assignment, int grade);
}
```

When new grade information is available (say, a new assignment is graded and entered, or an assignment is regraded and the old grade corrected), the grade database must communicate that information to the view. Let's suppose that Gill Bates has demanded a regrade on problem set 1, and that regrade did reveal grading errors: Gill's score should have been 30. The database code must somewhere make calls to `SpreadsheetView.update`. Suppose that it does so in the following way:

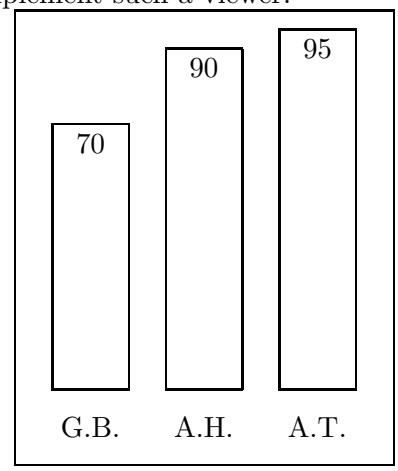
```
SpreadsheetView ssv = new SpreadsheetView();
...
ssv.update("6.170", "G. Bates", "PS1", 30);
```

(For brevity, this code shows literal values rather than variables for the `update` arguments.)

Then the spreadsheet view would redisplay itself in the following way:

| | PS1 | PS2 | PS3 |
|---|---|---|---|
| G. Bates | 30 | 85 | 80 |
| A. Hacker | 95 | 90 | 85 |
| A. Turing | 90 | 100 | 95 |

The staff might later decide that they would like to also view grade averages as a bargraph, and implement such a viewer:




Maintaining such a view in addition to the spreadsheet view requires modifying the database code:

```
SpreadsheetView ssv = new SpreadsheetView();
BargraphView bgv = new BargraphView();
...
ssv.update("6.170", "G. Bates", "PS1", 30);
bgv.update("6.170", "G. Bates", "PS1", 30);
```

Likewise, adding a pie chart view, or removing some view, would require yet more modifications to the database code. Object-oriented programming (not to mention good programming practice)

is supposed to provide relief from such hard-coded modifications: code should be reusable without editing and recompiling either the client or the implementation.

The observer pattern achieves the goal in this case. Rather than hard-coding which views to update, the database can maintain a list of observers which should be notified when its state changes.

```
Vector observers = new Vector();
...
for (int i=0; i<observers.size(); i++) {
  GradeDBViewer v = (GradeDBViewer) observers[i];
  v.update("6.170", "G. Bates", "PS1", 30);
}
```

In order to initialize the vector of observers, the database will provide two additional methods, `register` to add an observer and `remove` to remove an observer.

```
void register(GradeDBViewer observer) {
  observers.add(observer);
}

boolean remove(GradeDBViewer observer) {
  return observers.remove(observer);
}
```

The observer pattern permits client code (which manages the database and the viewers) to select which observers are active, and observers can even be added and removed at run-time.

This discussion has glossed over a number of details. For instance, the client might store all the information of interest to it (which might be all the 6.170 grades, or just the grades for some students, or just the number of updates to the database for a `DatabaseActivityViewer`), duplicating parts of the database, or the client might read the database when needed. A related design decision is whether the database sends all potentially relevant information to the client when an update occurs (this is the *push* structure), or the database simply informs the client, "an update has occurred" (this is the *pull* structure). The pull structure forces the client to request information, which may result in more messages, but overall a smaller amount of data transferred.

### 3.2 Blackboard

The blackboard pattern generalizes the observer pattern to permit multiple data sources as well as multiple viewers. It also has the effect of completely decoupling producers and consumers of information.

A blackboard is a repository of messages which is readable and writable by all processes. Whenever an event occurs that might be of interest to another party, the process responsible for or knowledgeable about the event adds to the blackboard an announcement of the event. Other processes can read the blackboard. In the typical case, they will ignore most of its contents, which do not concern them, but they may take action on other events. A process which posts an announcement to the blackboard has no idea whether zero, one, or many other processes are paying attention to its announcements.

Blackboards generally do not enforce a particular structure on their announcements, but a well-understood message format is required so that processes can interoperate. Some blackboards

provide filtering services so that clients do not see all announcements, just those of a particular type; other blackboards automatically send announcements to clients which have registered interest (this is a pull structure).

An ordinary bulletin board (either the physical or the electronic kind) is an example of a blackboard system. Another example of a blackboard at MIT is the zephyr messaging service.

The Liskov text calls this pattern "white board" rather than "blackboard." The former name may be more modern-seeming, but the latter is standard computer science terminology which has been in use for decades and will be more quickly recognized outside 6.170. The first major blackboard system was the Hearsay-II speech recognition system, implemented between 1971 and 1976.

# 4 Structural patterns

Wrappers modify the behavior of another class; they are usually a thin veneer over the encapsulated class, which does the real work. The wrapper may modify the interface, extend the behavior, or restrict access. The wrapper intermediates between two incompatible interfaces, translating calls between the interfaces. This permits two pieces of code that were not designed or written together, and thus are slightly incompatible, to be used together anyway.

Three varieties of wrappers are adapters, decorators, and proxies:

| Pattern | Functionality | Interface |
|---|---|---|
| Adapter | same | different |
| Decorator | different | same |
| Proxy | same | same |

The functionality and interfaces compared are those at the inside and outside of the wrapper; that is, a client's view of the wrapped object is compared to a client's view of the wrapper.

The remainder of this section discusses the three varieties of wrapper, then examine tradeoffs between two implementation strategies, subclassing and composition/forwarding.

## 4.1 Adapter

Adapters change the interface of a class without changing its basic functionality. For instance, they might permit interoperability between a geometry package that requires angles to be specified in radians and a client that expects to pass angles in degrees. Here are two other examples:

### 4.1.1 Example: Rectangle

Suppose that you have written code that works on `Rectangle` objects and calls their `scale` method.

```
interface Rectangle {
  // grow or shrink this by the given factor
  void scale(float factor);

  // other operations
  float area();
  float circumference();
  ...
}
```

```
class myClass {

  void myMethod(Rectangle r) {
    ...
    r.scale(2);
    ...
  }

}
```

Suppose there is another class `NonScaleableRectangle` which lacks the `scale` method but does have the other methods of `Rectangle`, as well as additional `setWidth` and `setHeight` methods.

```
class NonScaleableRectangle {
  void setWidth(float width) { ... }
  void setHeight(float height) { ... }
  ...
}
```

You may wish to switch to (or at least permit use of) this variety of rectangle, perhaps because it has desirable features, such as better performance, or perhaps because it is used elsewhere, in a system with which you need to interoperate.

You cannot use `NonScaleableRectangle` directly because of the incompatible interface. However, you can write an adapter which permits its use. There are two ways to do this: subclassing and composition/forwarding. The subclassing solution will be familiar:

```
class ScaleableRectangle1 extends NonScaleableRectangle implements Rectangle {
  void scale(float factor) {
    setWidth(factor * getWidth());
    setHeight(factor * getHeight());
  }
}
```

Composition/forwarding is a technique for "passing the buck", forwarding a request so that a different object does the requested work.

```
class ScaleableRectangle2 implements Rectangle {
  NonScaleableRectangle r;
  ScaleableRectangle2(NonScaleableRectangle r) {
    this.r = r;
  }

  void scale(float factor) {
    setWidth(factor * getWidth());
    setHeight(factor * getHeight());
  }

  float area() { return r.area(); }
```

```
  float circumference() { return r.circumference(); }
  ...
}
```

### 4.1.2 Example: Palette

Suppose that Professor Guttag calls Professor Devadas late at night because someone has discovered a problem with the problem set: it needs to support bicycles that can be repainted (to change their color). The professors split up the work: Professor Guttag will write a `ColorPalette` class with a method that, given a name like "red" or "blue" or "taupe", returns an array of three RGB values, and Professor Devadas will write code that uses this class. The professors do so, test their work, and go away for the weekend, leaving the the `.class` files for the TAs to integrate. They find that Professor Devadas has written code that depends on

```
interface ColorPalette {
  // returns RGB values
  int[] getColorPalette(String name);
}
```

but Professor Guttag has implemented a class that adheres to

```
interface ColorPallet {
  // returns RGB values
  int[] getColorPallet(String name);
}
```

What are the TAs to do? They do not have access to the source, and they do not have time to reimplement and retest. Their solution is to write a adapter for `ColorPallet` that changes the operation name. They can implement the adapter either by subclassing or by composition/forwarding, as described in Lecture 15.

## 4.2 Decorator

Whereas an adapter changes an interface without adding new functionality, a decorator extends functionality while maintaining the same interface. Typically, a decorator does not change existing functionality, only adds to it, so that objects of the resulting class behave exactly like the original ones, but also do something extra.

This sounds like subclassing, but not every instance of subclassing is a decoration. First, the implementation of an operation may be completely different or reimplemented in a subclass; that is not usually the case for a decorator, which contains relatively less functionality and reuses the superclass code. Second, subclasses can introduce new operations; wrappers (including decorators) generally do not.

An example of decoration is a `Window` interface (for a window manager) and a `BorderedWindow` interface. The `BorderedWindow` behaves exactly like the `Window`, except that it also draws a border around the outside.

Suppose that `Window` is implemented like this:

```
interface Window {
  // rectangle bounding the window
```

```
  Rectangle bounds();
  // draw this on the specified screen
  void draw(Screen s);
  ...
}

class WindowImpl implements Window {
  ...
}
```

The subclassing implementation would look like this:

```
class BorderedWindow1 extends WindowImpl {
  void draw(Screen s) {
    super.draw(s);
    bounds().draw(s);
  }
}
```

The composition/forwarding implementation would look like this:

```
class BorderedWindow2 implements Window {
  Window innerWindow;

  BorderedWindow2(Window innerWindow) {
    this.innerWindow = innerWindow;
  }

  void draw(Screen s) {
    innerWindow.draw(s);
    innerWindow.bounds().draw(s);
  }
}
```

### 4.3 Proxy

A proxy is a wrapper that has the same interface and the same functionality as the class it wraps. This does not sound very useful on the face of it. However, proxies serve an important purpose in controlling access to other objects. This is particularly valuable if those objects must be accessed in a stylized or complicated way.

For example, if an object is on a remote machine, then accessing it requires use of various network facilities. It is easier to create a local proxy that understands the network and performs the necessary operations, then returns the result. This simplifies the client by localizing network-specific code in another location.

As another example, an object may require locking if it can be accessed by multiple clients. The lock represents the right to read and/or update the object; without the lock, concurrent updates could leave the object in an inconsistent state, or reads in the middle of a sequence of updates could observe an inconsistent state. A proxy could take care of locking an object before an operation or sequence of operations, then unlocking it afterward. This is less error-prone than requiring clients to correctly implement the locking protocol.

Another variety of proxy is a security proxy. It might operate correctly if the caller has the correct credentials (such as a valid Kerberos certificate), but throw an error if an unauthorized user attempts to perform operations.

A final example is a proxy for an object that may not yet exist. If creating an object is expensive (because of computation or network latency), then it can be represented by a proxy instead. That proxy could immediately start to create the object in a background task in the hope that it is ready by the time the first operation is invoked, or it could delay creating the object until an operation is invoked. In the former case, the rest of the system can proceed without waiting; in the latter case, the work of creating the object need never be performed if it is never used. In either case, operations are delayed until the object is ready.

An example of a proxy for a non-existent object is Emacs's autoload functionality. For instance, I have a file `util-mde.el` which defines of a number of useful functions. However, I don't want to slow down Emacs by loading it every time I start Emacs. Instead, my `.emacs` file contains code like this:

```
(autoload 'looking-back-at "util-mde")
(autoload 'in-buffer "util-mde")
(autoload 'in-window "util-mde")
```

The form `(autoload 'function "file")` is essentially equivalent to (in Scheme syntax; Emacs Lisp uses `defun`)

```
(define function ()
  (load "file")  ;; redefine function
  (function)     ;; call the new version
)
```

Emacs autoloads most of its own functionality, from the mail and news readers to the Java editing mode. People who complain that Emacs starts up too slowly often have put indiscriminate `load` forms in their `.emacs` files; that's like using an inefficient implementation, then complaining that the compiler is poor because the resulting program runs slowly.

Proxy capabilities are particularly useful when clients have no knowledge of whether the object they are manipulating has special properties (such as being located on a remote machine, requiring locking or security, or not being loaded). It is best to insulate the client from such concerns and localize them in a proxy wrapper.