LEVELIZATION TECHNIQUES:

- Escalation: Moving mutually dependent functionality higher in the physical hierarchy. (p. 215)
- Demotion: Moving common functionality lower in the physical hierarchy. (p. 229)
- Opaque Pointers: Having an object use another in name only. (p. 247)
- Dumb Data: Using data that indicates a dependency on a peer object, but only in the context of a separate, higher-level object. (p. 257)
- Redundancy: Deliberately avoiding reuse by repeating small amounts of code or data to avoid coupling. (p. 269)
- Callbacks: Using client-supplied functions that enable lower-level subsystems to perform specific tasks in a more global context. (p. 275)
- Manager Class: Establishing a class that owns and coordinates lower-level objects. (p. 288)
- Factoring: Moving independently testable subbehavior out of the implementation of complex components involved in excessive physical coupling. (p. 294)
- Escalating Encapsulation: Moving the point at which implementation details are hidden from clients to a higher level in the physical hierarchy. (p. 312)

INCREMENTAL INSULATION TECHNIQUES:

- Removing private inheritance by converting WasA to HoldsA. (p. 349)
- Removing embedded data members by converting HasA to HoldsA. (p. 352)
- Removing private member functions by making them static at file scope and moving them to the .c file. (p. 353)
- Removing protected member functions by creating a separate utility component and/or extracting a protocol. (p. 363)
- Removing private member data by extracting a protocol and/or moving static data to the .c file at file scope. (p. 375)
- Removing compiler-generated functions by explicitly defining these functions. (p. 378)
- Removing include directives by removing unnecessary include directives or replacing them with (forward) class declarations. (p. 379)
- Removing default arguments by replacing valid default values with invalid default values or employing multiple function declarations. (p. 381)
- Removing enumerations by relocating them to the .c file, replacing them with const static class member data, or redistributing them among the classes that use them. (p. 382)

TOTAL INSULATION TECHNIQUES:

- Protocol Class: Creating an abstract “protocol” class is a general insulation technique for factoring the interface and implementation of an abstract base class. Not only are clients insulated from changes to the implementation at compile time, but even link-time dependency on a specific implementation is eliminated. (p. 386)
- Fully Insulating Concrete Class: A “fully insulating” concrete class holds a single opaque pointer to a private structure defined entirely in the .c file. This struct contains all of the implementation details that were formerly in the private section of the original class. (p. 398)
- Insulating Wrapper Component: The concept of an encapsulating wrapper component (from Chapter 5) can be extended to a fully insulating wrapper component. Wrappers are typically used to insulate several other components or even an entire subsystem. Unlike a procedural interface, a wrapper layer requires considerable up-front planning and top-down design. In particular, care must be taken in the design of a multi-component wrapper to avoid the need for long-distance friendships. (p. 405)
DEFINITION:

NOTATION

X
x

MEANING

X is a logical entity (e.g., class).

x is a physical entity (e.g., file).

B \text{IsA} \rightarrow A

B uses A in B's interface.

B \text{Uses-In-The-Interface} \rightarrow A

B uses A in B's implementation.

B \bullet \text{Uses-In-The-Implementation} \rightarrow A

class Car {
    // ...
};

// car.c
#include "car.h"
// ...

class Car : public Vehicle {
    // ...
};

class Car {
    // ...
    public:
    void addFuel(Gas *);
    // ...
};

class Car {
    Engine d_motor;
    // ...
};

Our Notation

Uses In The Interface

Uses In The Interface
(In Name Only)

Uses In The Implementation

Booch Notation

Uses In The Interface

HasA/HoldsA
(Unspecified)

HoldsA
(By Reference)

HasA
(By Value)
DEFINITIONS:

A type is *used in the interface* of a function if the type is referred to when declaring that function. (p. 50)

A type is *used in the (public) interface* of a class if the type is used in the interface of any (public) *member* function of that class. (p. 51)

A type is *used in the implementation* of a function if the type is referred to in the definition of that function. (p. 53)

A type is *used in the implementation* of a class if that type (1) is used in a member function of the class, (2) is referred to in the declaration of a data member of the class, or (3) is a private base class of the class. (p. 55)

Specific kinds of the Uses-In-The-Implementation Relationship: (p. 55)

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses</td>
<td>The class has a member function that names the type.</td>
</tr>
<tr>
<td>HasA</td>
<td>The class embeds an instance of the type.</td>
</tr>
<tr>
<td>HoldsA</td>
<td>The class embeds a pointer (or reference) to the type.</td>
</tr>
<tr>
<td>WasA</td>
<td>The class privately inherits from the type.</td>
</tr>
</tbody>
</table>

A class is *layered* on a type if the class uses that type substantively in its implementation. (p. 58)

A component *y DependsOn* a component *x* if *x* is needed in order to compile or link *y*. (p. 121)

A component *c* uses a type *T in size* if compiling *c* requires having first seen the definition of *T*. (p. 248)

A component *c* uses a type *T in name only* if compiling *c* and any of the components on which *c* may depend does not require having first seen the definition of *T*. (p. 249)