Object Oriented Design Principles

Jim Fawcett

Spring 2017

Based on a series of articles by Robert Martin,

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Contents

Bad Designs

- What makes a design bad? Robert Martin suggests:
  - ***Rigidity***
    It is hard to change because every change affects too many other parts of the system.
  - ***Fragility***
    When you make a change, unexpected parts of the system break.
  - ***Immobility***
    It is hard to reuse in another application because it cannot be disentangled from the current application.

- The design principles discussed in the following are all aimed at preventing “bad” design.

The Open/Closed Principle

based on an article of that title by Robert Martin

Statement of Principle

- Software entities (classes, packages, functions) should be open for extension, but closed for modification.

- Definitions
  - Open:
    A component is open if it is available for extension:
    - add data members and operations through inheritance.
    - Create a new policy template argument for a class that accepts policies.
  - Closed:
    A component is closed if it is available for use by other components but may not, itself, be changed, e.g., by putting it under configuration management, allowing read only access.

Origin and Motivation

- The open/closed principle was stated and discussed as one of the fundamental object oriented principles by Bertrand Meyer in:
  Object Oriented Software Construction, Prentice-Hall, 1988.

- In large complex systems, changes occur often, due to:
  - changing requirements
  - latent errors
  - performance issues

  Making changes can often result in cascades of changes to other, dependent components.

- The open/closed principle says that components should never change, only be extended to meet changing requirements.

The Role of Abstraction

- It is very difficult to build components that don't change, but only support extension:
  - Latent errors force change. We can't fix incorrect operation by extension. The component itself must be fixed.
  - Performance failures force change. When performance needs are not met we are forced to change the implementation to perform better, usually by changing a computational algorithm or data structure.

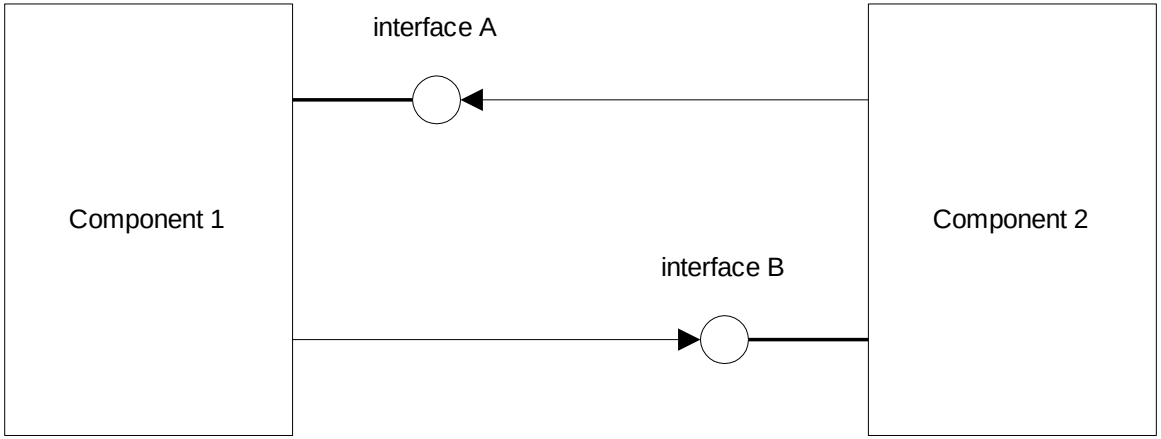
- What we can do is represent the component by an abstract interface, e.g., an abstract class, which provides a protocol that derived classes implement.

Abstract Interfaces

- When we program to abstract interfaces:
  - changes to derived classes which implement the interface will not break any client code, and may not even require recompilation of some clients.
  - What we can't do is change the interface definition. Any change here may force changes on most or all of its clients.

- Abstract interfaces directly support the Open/Closed Principle. They must be extended, but are closed to modification. Since they have no implementation they have no latent errors to fix and no performance issues.

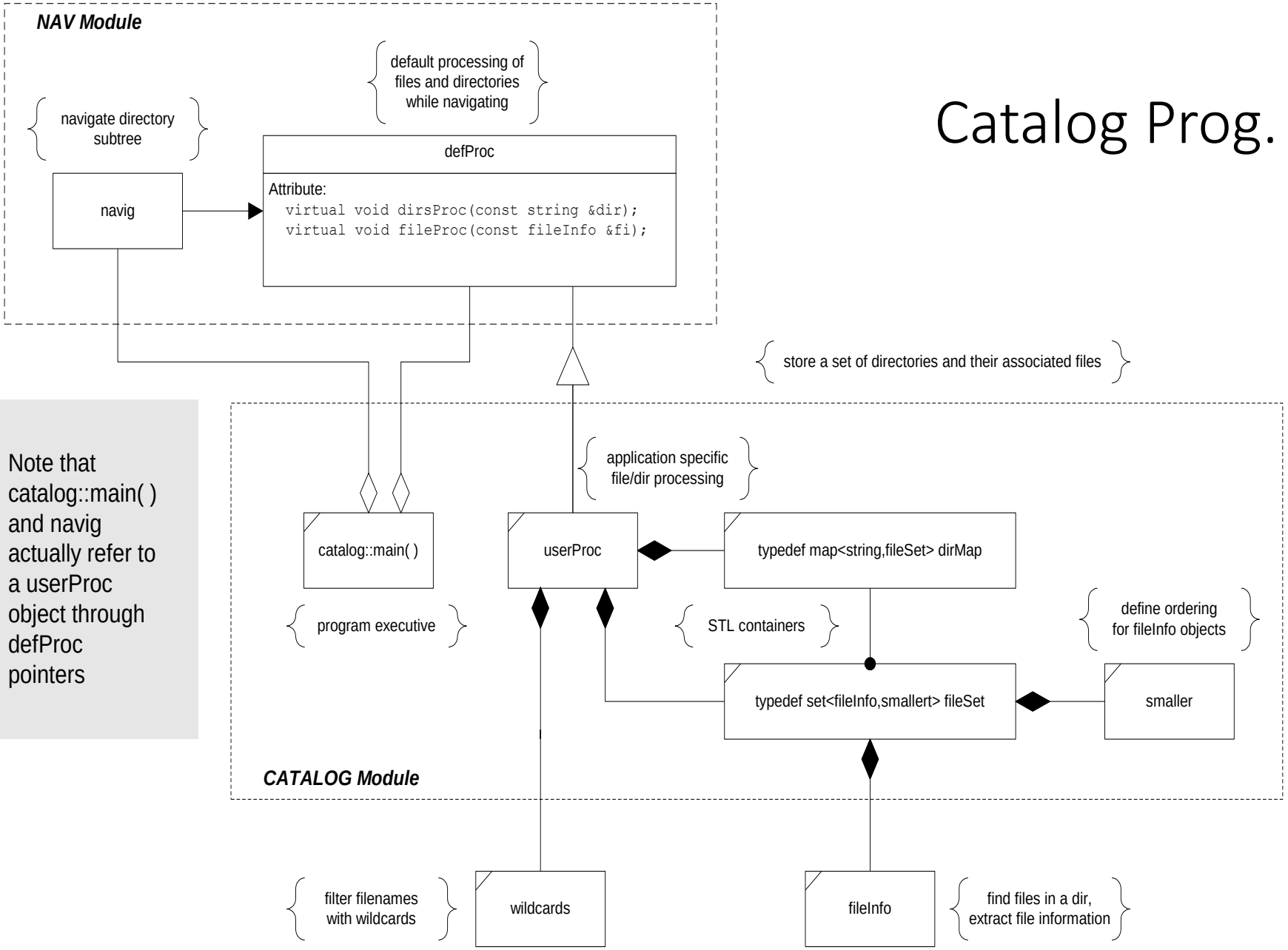
Abstract Interfaces



A Real Example

- Recall the catalog program we've discussed several times in class. Its class diagram is shown on the next page.
  - Since we want to reuse the navig class in many programs without change it is important that the defProc class satisfy the Open/Closed Principle. If defProc doesn't change then navig can stay the same.
  - But, since navig can know nothing of the application details of a program designed after it was released, we depend on extensions of defProc to supply the needed application program processing.
  - defProc is not an abstract base class in strict C++ terms since it provides default processing. However, it behaves like an abstract class in that it provides a protocol for navig to use and expects derived classes to override its virtual methods.

Catalog Prog.



Note that catalog::main( ) and navig actually refer to a userProc object through defProc pointers

A Less Radical Approach

- The open/closed principle says that components should never change, only be extended to meet changing requirements.

- Robert Martin, in his paper “The Open-Closed Principle” does not propose quite so drastic a design paradigm. He suggests:
  - Make all member data private, e.g., no public, no protected data.
  - No global data - ever.
  - No use of RTTI
  - Use polymorphism and/or templates to provide extensions

- Few would suggest that 100 percent of every design should satisfy the open/closed principle:
  - An effective design may use many components which do satisfy the principle but also include program “glue”.
  - The glue is used to bind the components into a working program, without much regard to the open/closedness of the glue part.

Summary

Table of Contents

- The Open/Closed Principle states that:
  - well designed code can be extended without modification
  - new features are added by adding new code rather than changing already working code

- Software that is designed to be reusable, maintainable, and robust must be extensible without requiring change.
  - We do this with abstract classes
  - algorithms make use of virtual functions
  - they are extended by derived classes that implement the virtual functions in different ways

The Liskov Substitution Principle

based on an article of that title by Robert Martin

Ref: Barbara Liskov, “Data Abstraction and Hierarchy”, SIGPLAN Notices, 23, 5 (May 1988)

Statement of Principle

- Functions that use pointers or references statically typed to some base class must be able to use objects of classes derived from the base through those pointers or references without any knowledge specialized to the derived classes.

- We have seen how powerful this principle is in helping us design loosely coupled systems. The base class provides a protocol for clients to use regardless of what derived class is receiving the client's messages.

Substitution Failures

- A hierarchy of classes will fail to satisfy this principle if any of the following are true:
  - The base class does not make its destructor virtual
  - Derived classes redefine non-virtual member functions of the base
  - Virtual functions are overloaded or given default parameters
  - Clients use dynamic_cast to access derived class extensions to base class protocol through base class pointers or references.

More Subtle Failures

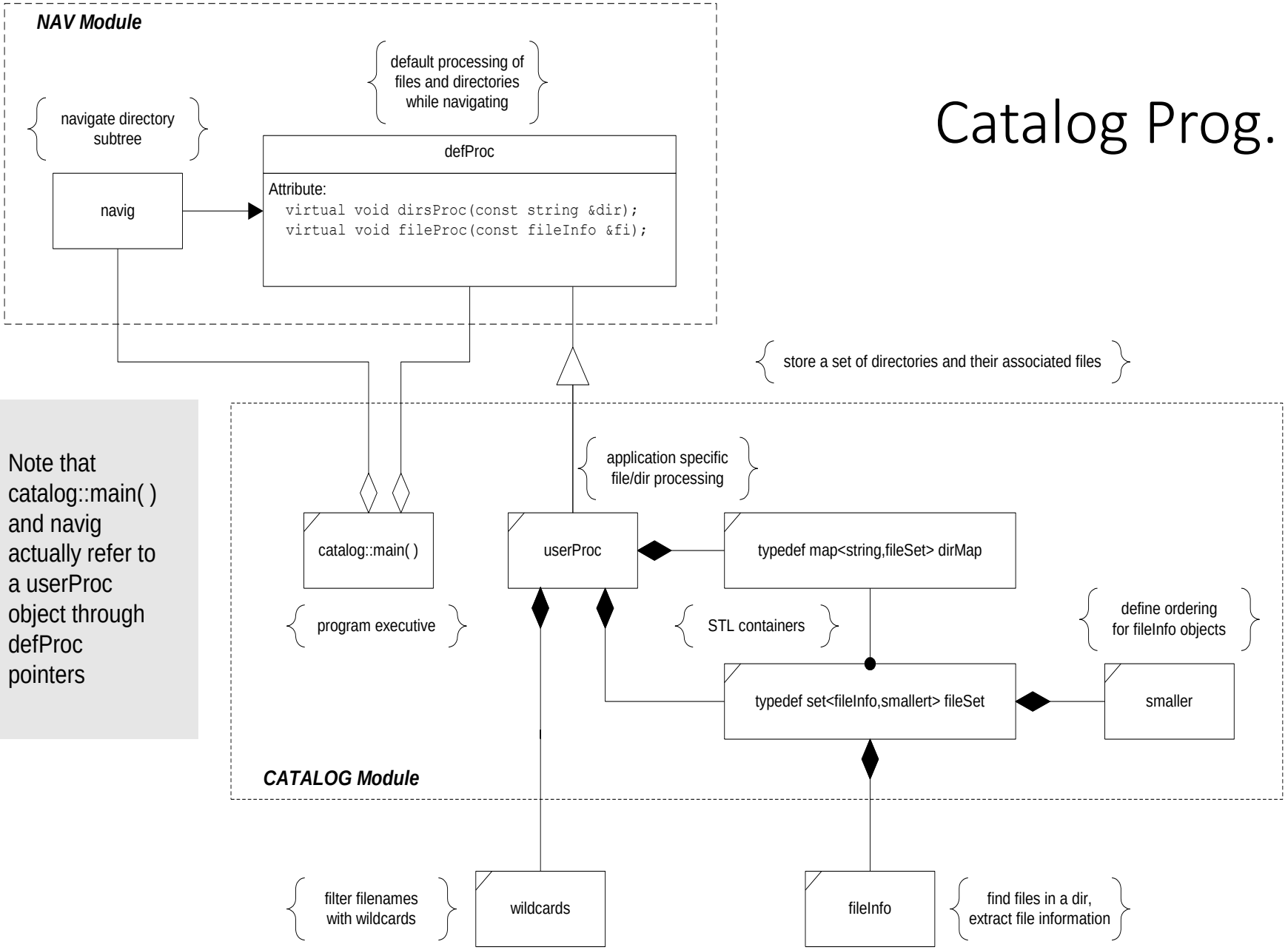
- Martin points out that substitution failures can happen for more subtle reasons.
  - Deriving a square from a rectangle implies that one of the state variables, height or width, is redundant. Clients of rectangle need to know they are working with square if they take advantage of square's property - height = width.

- The Liskov Substitution Principle implies that "is-a" relationships are based on behavior, not some intrinsic mental model.
  - The behavior of a square - change its height and you change its width - does not apply to rectangles and so square objects are not rectangle objects.

A Real Example

- Consider again the catalog program. It is crutial for the nav module's reuse that the navig class uses no knowledge of the application program.
  - nav is a very useful facility and we want to use it in many designs, but only maintain one component.
  - It is critically important that the navig class depends only on its defProc interface and not on implementation details of the catalog data structure.
  - If navig is to remain unchanged in various applications it must be able to use its base class defProc pointer with no knowledge of the specifics of the catalog's userProc classes, e.g., navig's use of the defProc hierarchy satisfies the Liskov Substitution Principle.

Catalog Prog.



Note that catalog::main( ) and navig actually refer to a userProc object through defProc pointers

Summary

Table of Contents

- The Liskov Substitution Principle states that every function that operates on a base class reference or pointer should be able to operate successfully when a derived class object is substituted for the base object. In doing this it should need no information about the derived object, or even know that the object is not a base class instance.

- To ensure a design supports the Liskov Substitution Principle:
  - derived objects must not expect users to obey pre-conditions stronger than expected for the base class
    - their pre-conditions must be no stronger
  - derived objects must satisfy all of the post-conditions satisfied by the base class
    - their post-conditions must be no weaker
  - base classes must provide virtual functions including a virtual destructor.

Dependency Inversion Principle

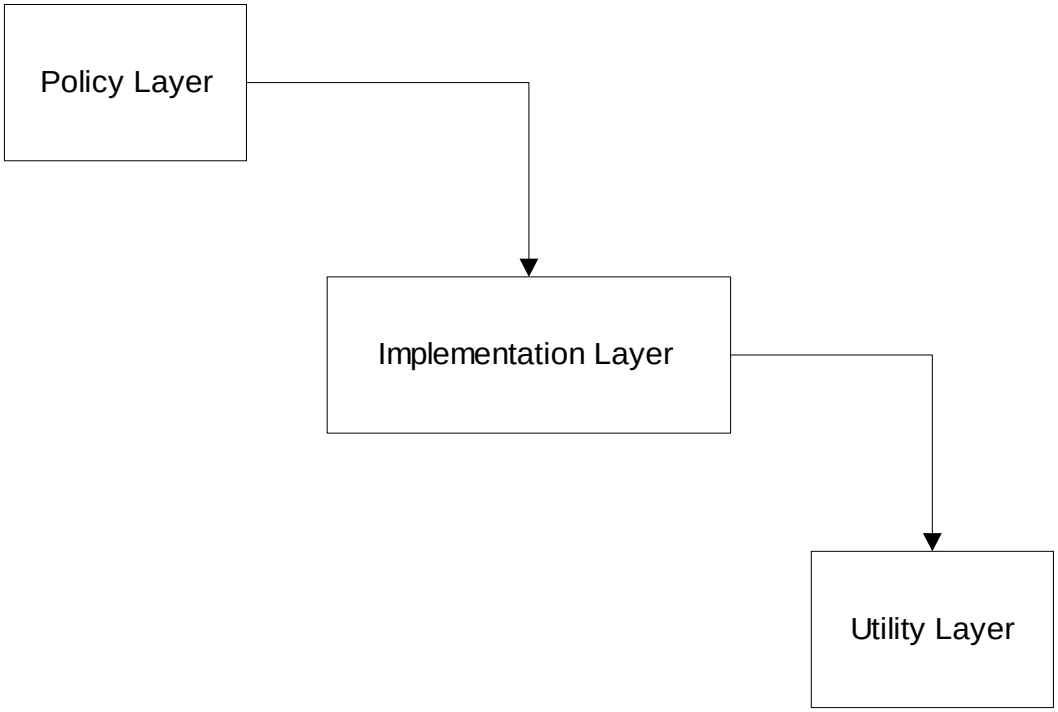
based on an article of that title by Robert Martin

Statement of Principle

- Dependency Inversion Principle:
  - High level components should not depend upon low level components. Instead, both should depend on abstractions.
  - Abstractions should not depend upon details. Details should depend upon the abstractions.

- We all can agree that complex systems need to be structured into layers. But if that is not done carefully the top levels tend to depend on the lower levels.
  - On the next page we show a “standard” architecture that appears to be practical and useful.
  - Unfortunately it has the ugly property that policy layer depends on implementation layer which depends on utility layer, e.g., dependencies all the way down.

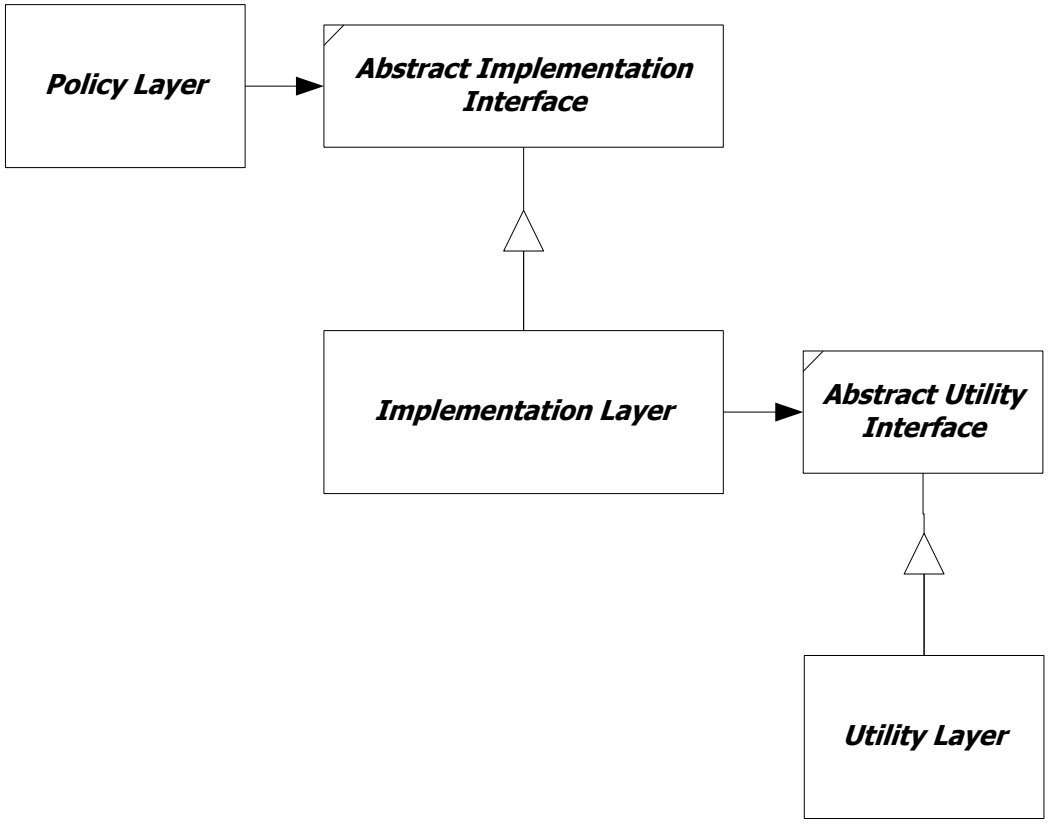
A Layered Architecture



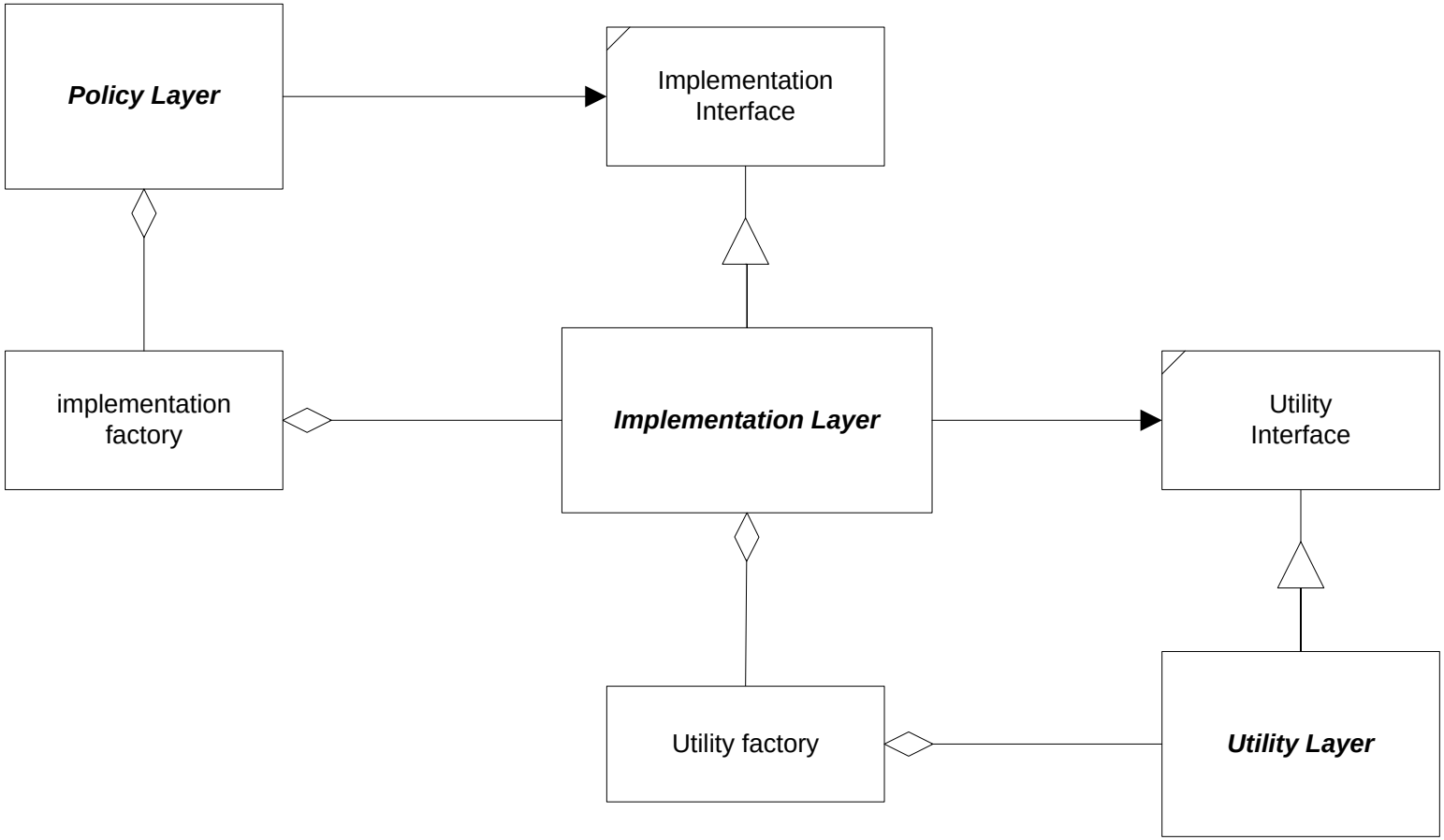
Using Abstract Layers

- The diagram on the next page shows a “better” model, e.g., less rigid, less fragile, more mobile.
  - Each layer is separated by an abstract interface.
    - policy depends, not on the implementation, but only on its abstract interface.
    - implementation depends only on its interface and on the interface defined by utility
    - utility depends only on its published interface
  - Policy is unaffected by any changes to implementation and utility and implementation is unaffected by changes to utility.
    - as long as we transport the interface along with its component each of the three components is reusable and robust

Layers Using Abstraction



Decouple using Object Factories

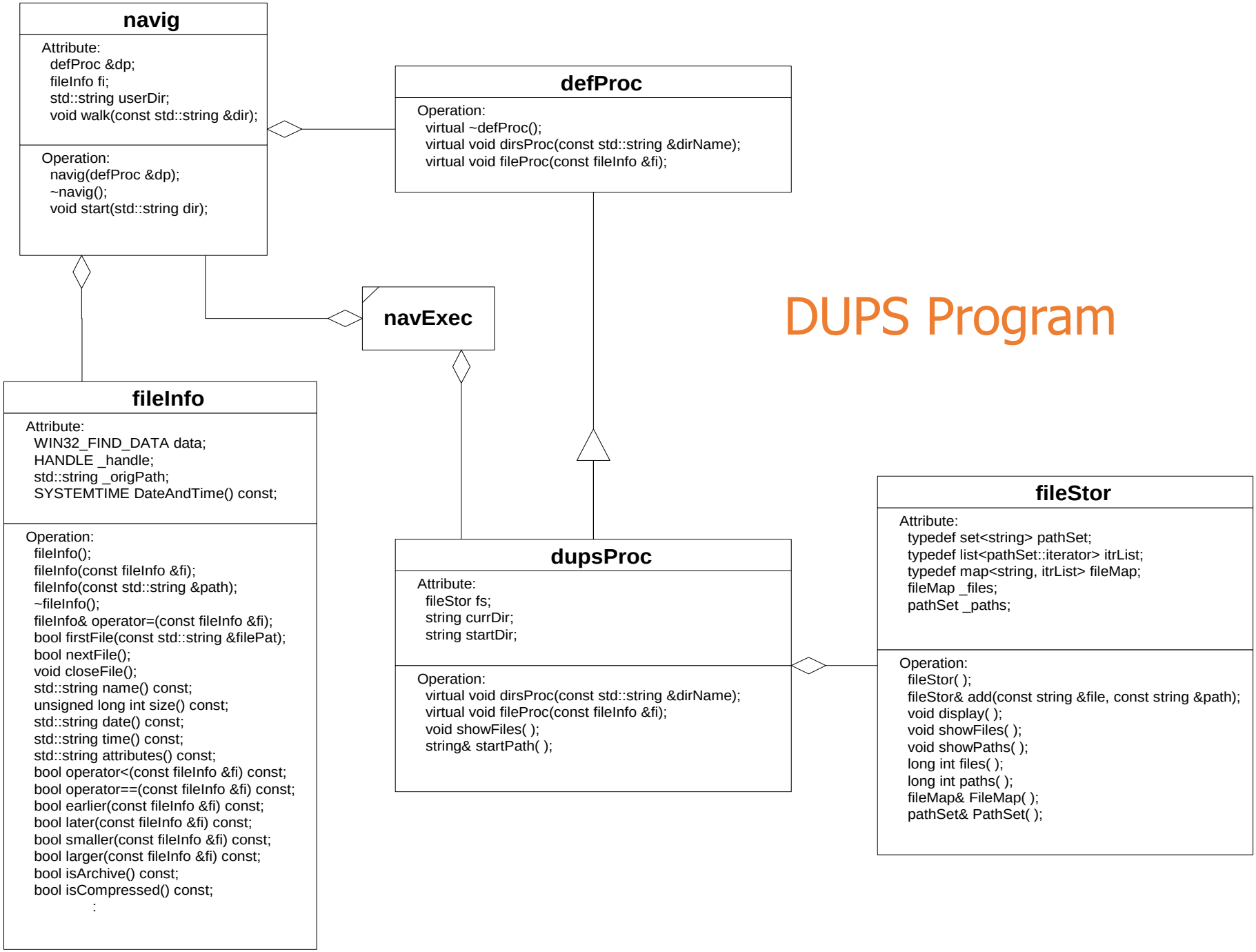


A Real Example

- The dups program, discussed earlier in class illustrates the dependency inversion principle. In fact it has the “layered abstraction” structure shown on the previous page. Its function is to find all files with duplicate names on some directory subtree.

  - Nav module communicates only to the abstract defProc interface. Nav is where the policy for searching is established.

  - Dups fileStore implementation depends on the STL utilities, but only on their published “abstract” interfaces. The designer of the fileStore module can substitute different allocator objects in the STL containers or, if hash-based set and map containers were available (there are some open-software implementations available that adhere to the STL interfaces) they could be used without affecting the rest of the design in any way except to make it faster.


navig
Attribute:
defProc &dp;
fileInfo fi;
std::string userDir;
void walk(const std::string &dir);
Operation:
navig(defProc &dp);
~navig();
void start(std::string dir);
defProc
Operation:
virtual ~defProc();
virtual void dirsProc(const std::string &dirName);
virtual void fileProc(const fileInfo &fi);
navExec
DUPS Program
fileInfo
Attribute:
WIN32_FIND_DATA data;
HANDLE _handle;
std::string _origPath;
SYSTEMTIME DateAndTime() const;
Operation:
fileInfo();
fileInfo(const fileInfo &fi);
fileInfo(const std::string &path);
~fileInfo();
fileInfo& operator=(const fileInfo &fi);
bool firstFile(const std::string &filePat);
bool nextFile();
void closeFile();
std::string name() const;
unsigned long int size() const;
std::string date() const;
std::string time() const;
std::string attributes() const;
bool operator<(const fileInfo &fi) const;
bool operator==(const fileInfo &fi) const;
bool earlier(const fileInfo &fi) const;
bool later(const fileInfo &fi) const;
bool smaller(const fileInfo &fi) const;
bool larger(const fileInfo &fi) const;
bool isArchive() const;
bool isCompressed() const;
:
dupsProc
Attribute:
fileStor fs;
string currDir;
string startDir;
Operation:
virtual void dirsProc(const std::string &dirName);
virtual void fileProc(const fileInfo &fi);
void showFiles( );
string& startPath( );
fileStor
Attribute:
typedef set<string> pathSet;
typedef list<pathSet::iterator> itrList;
typedef map<string, itrList> fileMap;
fileMap _files;
pathSet _paths;
Operation:
fileStor( );
fileStor& add(const string &file, const string &path);
void display( );
void showFiles( );
void showPaths( );
long int files( );
long int paths( );
fileMap& FileMap( );
pathSet& PathSet( );

Summary

Table of Contents

- The Dependency Inversion Principle states that components that encapsulate high level policy should not depend on components that implement details.

- Instead, both kinds of components should depend on abstractions.

Interface Segregation Principle

based on an article of that name by Robert Martin

Statement of Principle

- Clients should not be forced to depend upon interfaces they do not use.
  - this applies to clients of the public interface of a class
  - it also applies to derived classes
- We create interfaces to satisfy the needs of clients. When a component has several different clients it is tempting to provide a large interface that satisfies the needs of all clients.
- It is much better design to have the component support multiple interfaces, one appropriate for each client.
  - Otherwise, if we have to change an interface we affect even those clients that do not use the features we change.

Motivation

- The earlier principles say that we should never change an interface, and that one way to help keep interfaces immutable is to make them abstract.

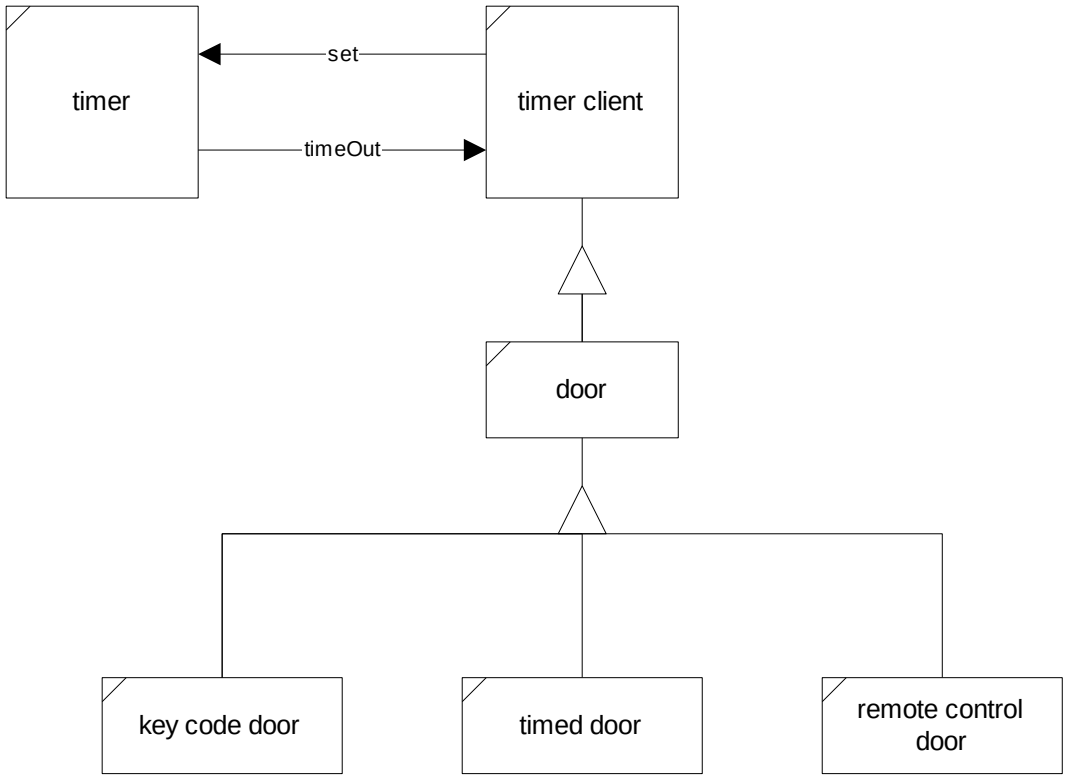
- However, the reality is that not all interfaces can, or even should be abstract. When we have such a situation it is important to keep the impact of changing an interface localized.

- The intent of this principle is that we push interfaces down the decomposition hierarchy to the clients that really need them.

An Example

- The diagram on next page shows a class hierarchy designed to support implementation of electronically secured doors. The base door class provides support for timed doors, key code doors, and remote control doors.
  - Base door class is doing too much. Here it must supply default behaviors for each of the three types of doors which do nothing since each behavior is needed by only one of the derived classes.
  - The problem is that any change in one behavior will cause recompilation of all derived classes. Suppose we ship a revised door component in the form of a new dynamic link library. Every client will have to be recompiled. They can't just use the new interface.
  - Even making base door class abstract, which avoids some of these problems, clients still have to deal with a complicated interface.

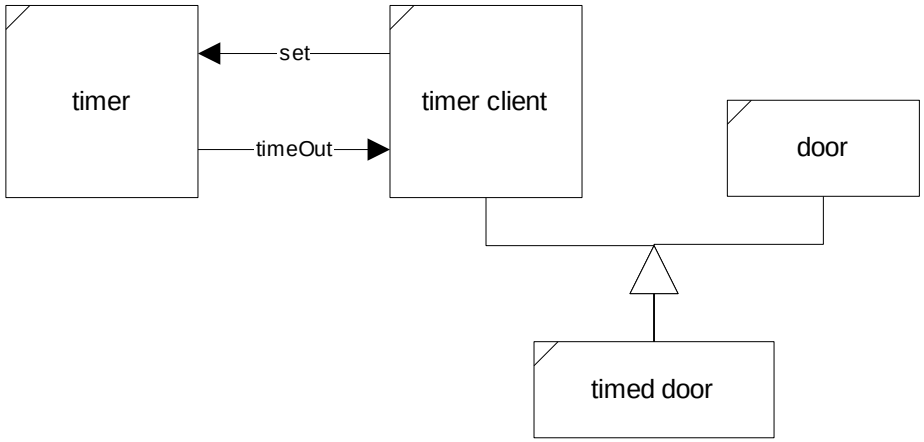
Fat Interfaces

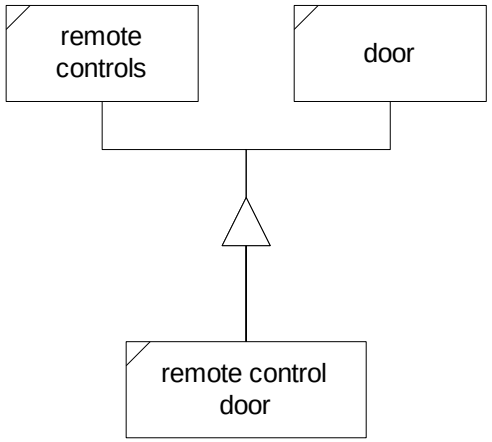


Segregating Interfaces

- The fat door interface can be slimmed down by using multiple inheritance, as shown in the next diagram.
  - The door base class now only provides those behaviors common to all doors.
  - Each of the specialized activities is segregated into its own base class, e.g., key code base and remote control base
- Now the door class provides only the behaviors for doors and doesn't need to provide any null behaviors.
- When an interface changes, only the clients that use the interface will need to be recompiled.

Putting Door Interface on a Diet





Summary

Table of Contents

- The Interface Segregation Principle states that:
  - fat interfaces lead to inadvertent couplings between clients that ought to be isolated
  - fat interfaces can be segregated, through multiple inheritance, into abstract base classes that break unwanted coupling between components.
  - clients simply mix-in the appropriate interfaces for their activities.

The Granularity Issue

based on an article by Robert Martin

Partitioning into Packages

- As software becomes large and complex we need to enforce some form of partitioning that is larger than the class and smaller than a program.

- Packages represent a grouping of classes into a cohesive structure that represents a single high-level abstraction.

- Packages allow us to reason about and reuse software on a large scale without being swamped with detail.

Reuse/Release Equivalence Principle

- The granule of reuse is the granule of release. Only components that are released through a tracking system can be effectively reused.
  - We reuse code if, and only if, to use you don't have to look at source code, only its public header files. You need only link with static librar-ies or include dynamic link libraries.
  - Whenever these libraries are fixed or enhanced we receive new versions which we install at our convenience.
- This granule is the package.
  - Is a package a module? Not necessarily. A package can be one or more modules - the important thing is that for tracking they are treated as a unit.

Common Reuse Principle

- The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.
  - Generally, reusable classes collaborate with other classes that are part of the reusable abstraction. This principle states that these classes belong in a package.
  - When a package is reused, a dependency is created on the whole package.
  - Every time the package is released the applications that reuse it must be re-evaluated and re-released.
  - We want to make sure that when we depend on a package we depend on most of the classes in the package, otherwise we are re-evaluating and re-distributing more than is necessary and wasting effort (remember “Small is Beautiful”).

Common Closure Principle

- The classes in a package should be closed together against the same kinds of changes. A change that effects a package affects all the classes in that package.
  - If code in an application must change (and that happens constantly during development) we want the changes to occur entirely in one package.
  - If we fix a latent error or performance bug we want to fix and re-release only one package.
  - The Common Closure Principle is an attempt to gather together in one place all the classes that are likely to change for the same reasons.
  - The principle groups together classes which cannot be closed against certain types of changes, e.g., based on requirements or platform.

Acyclic Dependencies Principle

- The dependency graph describing relationships between packages in a program must contain no cycles.
  - The issue here is to ensure there are no mutual dependencies.
  - The package is a unit of work assigned to an individual or team.
  - When a package is released, it is put under configuration control and made available for others to use.
  - Other teams can decide whether or not to immediately adopt the new release, since adopting a new release may break their code.
  - Thus changes made do not need to have an immediate effect on another team.
  - To make this sensible process work there can be no mutual dependencies between packages.

Build Map

- The package dependency graph is a build map.
  - lowest level packages are built first
  - then the packages that depend of them are build
  - this process continues until the build is complete

- Packages may or may not have a structure that matches the activity decomposition in the program.
  - that is why packages are not the same as modules
  - a program is described by its activities
  - activities are designed and implemented with modules
  - modules are partitioned into packages to isolate mutual dependencies to within a single package.
  - packages are the unit of assigned work and release

Summary

Table of Contents

- The Granularity Issue is one of implementation more than design. There are three important principles associated with a system's granularity:
  - The Reuse/Release Equivalence Principle - packages are the unit of release.
  - The Common Reuse Principle - classes in a package are reused together.
  - The Common Closure Principle - classes in a package should be closed together against the same kinds of changes.
  - The Acyclic Dependencies Principle - the dependencies between packages must have no cycles.

The Stability Issue

based on an article by Robert Martin

Volatile & Nonvolatile Dependencies

- Volatility describes how frequently a component changes.
  - if we say a component is volatile we mean that it is likely to change, perhaps due to changing requirements, latent errors, performance issues, or changes of platform
  - if we say a component is non-volatile we mean that it is very unlikely to change.

- Stability measures how volatile the component is:
  - The fewer things a component depends upon the more stable it is. An abstract interface is highly stable.
  - The harder it is to change the more stable it is. A common utility that many components already use is highly stable.

Independence and Responsibility

- Independent classes are classes which do not depend on anything else.
  - Abstract classes are nearly independent - they do depend on the needs of their clients.
  - Independent classes are stable. Nothing drives their change.

- Responsible classes are classes that are heavily depended upon.
  - A common utility like the standard C++ STL is responsible.
  - Responsible classes are stable. They are too hard to change because a change implies many related changes in clients.

- The most stable classes are both Independent and Responsible.

Stable Dependencies Principle

- Every complex program must be layered to make its implementation manageable. We organize these layers into packages. This implies dependency relationships between the packages that make up a program.

- The Stable Dependencies Principle states that:
  - the dependencies between packages in a design should be in the direction of stability of the packages
  - a package should only depend on packages that are more stable than it is

Good Volatility

- Some volatility is necessary in a design if it is to be maintained.
  - We encourage volatility with the Open/Closed Principle.
  - By using this principle we design packages to support certain kinds of changes.
- Any package that is difficult to change should not depend on a package that we expect to be volatile.

Positional Stability
Measuring Dependency

- Positional stability is based on the number of dependencies that enter and leave a package:
  - Afferent Couplings:
    Ca = number of classes outside package depending on classes inside package
  - Efferent Couplings:
    Ce = number of classes inside package that depend on outside classes
  - Instability:
    $I = Ce/(Ca + Ce)$

    $I \in [0,1]$, $I=0 \Rightarrow$ maximum stability, $I=1 \Rightarrow$ minimum stability

- If we are careful only to #include files that we depend on and we isolate one class per file, then we can compute I by counting includes.

Stable Abstractions Principle

- The abstraction of a package should be proportional to its stability:
  - Packages that are maximally stable should be maximally abstract.
  - Unstable packages should be concrete.

- If a package is to be stable it should expose abstract interfaces so that it can be extended.
  - stable packages that are extensible are flexible and do not constrain the design

- The Stable Abstractions Principle combined with the Stable Dependencies Principle amount to a Dependency Inversion Principle for packages.

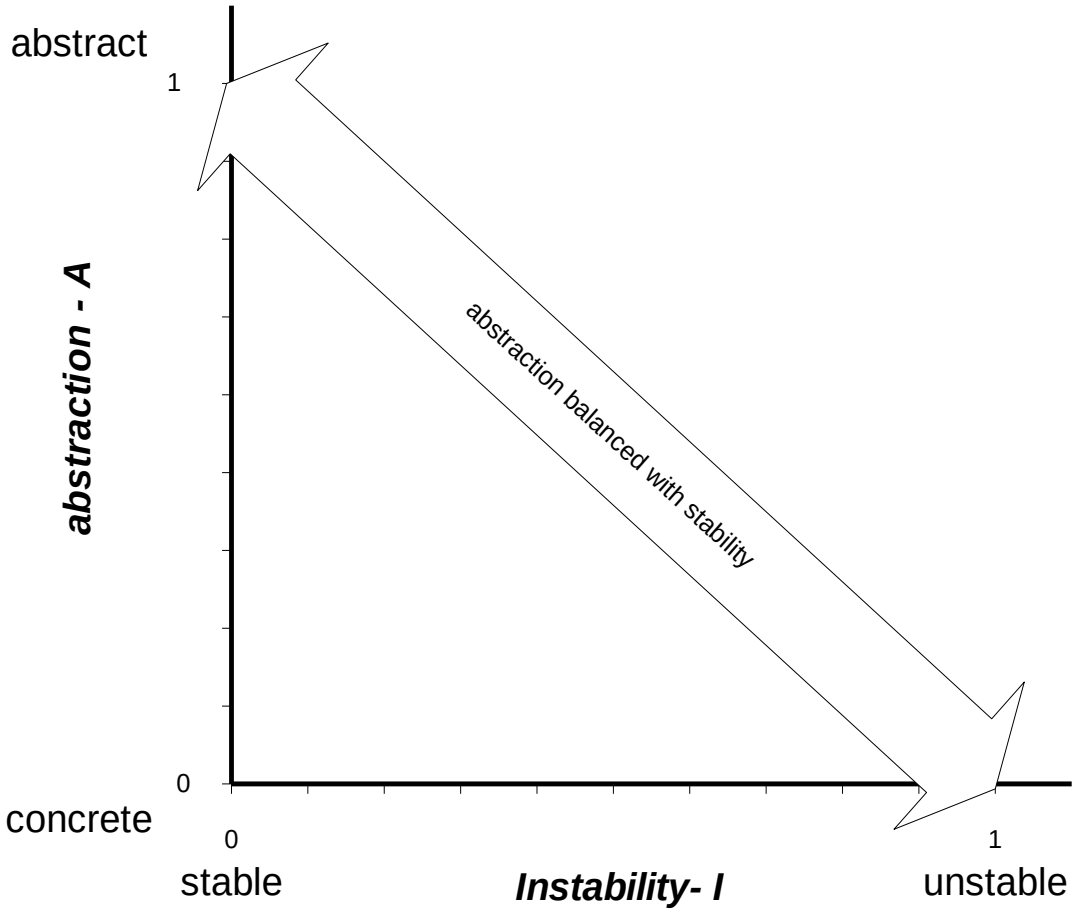
Measuring Abstraction

- A measure of abstraction is the ratio of the number of abstract classes to total number of classes:

  A = number of abstract classes / total number of classes

  $A \in [0,1]$.
  $A = 1 \Rightarrow$ maximum abstraction, $A = 0 \Rightarrow$ minimum abstraction

Stable Abstractions Principle



Distance Metric

- The distance of a design from the balanced state is measured by:

  $$D = A + I - 1$$

  This is the perpendicular distance of the design from the balanced line on the previous chart.

Summary

Table of Contents

- The Stability Issue is concerned with volatility of components.
  - we require stability of responsible components
  - we expect stability of independent components
- Not all instability is bad.
  - A system has to be mutable in order that latent errors and performance failures can be fixed.
  - A system has to be mutable in order to satisfy changing or new requirements
- A good design develops packages that maintain a balance between abstractness and instability as measured by D.

End of Presentation