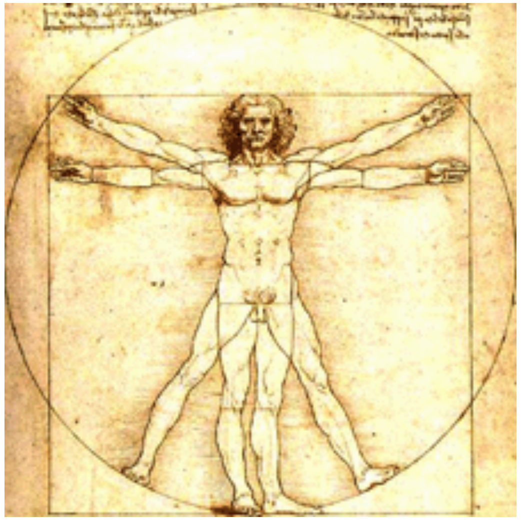
Design Principles



❖ ***Class Design***

- Open-Closed Principle (OCP)
- Don't Repeat Yourself Principle (DRY)
- Design by Contract
- Single Responsibility (SRP)
- Liskov Substitution (LSP)
- Dependency Inversion (DIP)
- Interface Segregation (ISP)

❖ ***Package Coupling***

- Acyclic Dependencies
- Stable Dependencies
- Stable Abstractions

❖ ***Package Cohesion***

- Reuse-Release Equivalence
- Common Closure
- Common Reuse

Open-Closed Principle (OCP)

- *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.*
  - *Bertrand Meyer "Object Oriented Software Construction"*
- In an ideal world, you *should never need to change existing code or classes*.
  - New functionality can be added by adding new subclasses and overriding methods, or by reusing existing code through delegation.
- Prevents introduction of new bugs in existing code.
  - If you never change it, you can't break it.
- *Create abstractions that are fixed, yet represent an unbounded group of possible behaviours*.
  - Abstractions are ***base classes***, unbound groups are ***derived***.
  - *Using inheritance, we can create derived classes that conform to the abstract polymorphic interfaces defined in abstract base classes.*
- OCP concerns ***flexibility*** and goes beyond inheritance.
  - Really a combination of abstraction and encapsulation.

Don't Repeat Yourself Principle

- *Avoid duplicate code by abstracting out things that are common and placing those things in a single location.*
  - Sounds simple, but DRY principle is critical in writing code that is easy to maintain and reuse.
  - ***Nothing new*** about avoiding duplicate code. Subroutines/ functions invented for this reason.
  - Object-oriented design allows us to approach the problem in a new way. Requires holistic approach, ***requirements→coding***.
- *DRY implies that each piece of information and behaviour (one requirement)* ***should be in a single, sensible place.***
  - To avoid duplicate code, only implement each feature and requirement in an application ***one single time***.
  - Begin by abstracting out duplicate code.
- Should be part of the ***requirements gathering*** process.
  - A requirement should be implemented only once. Use-cases shouldn't have to overlap.

Single Responsibility Principle

- *Every object in a system should have a single responsibility, and all the object's services should be focused on carrying out that single responsibility.*
  - SRP is implemented correctly when each object has ***only one reason to change***.
- ***Responsibility:*** a *reason* for change. If there is more than one motive for changing a class, that class has more than one responsibility.
  - Related to DRY principle in that no other classes should share a behaviour because no other classes should share responsibility.
- For example, consider a Rectangle class with *area()* and *draw()* methods.
  - Implies that all applications that wish to compute an area also want to draw. Two responsibilities have been mixed into one class...
  - A class's responsibility is ***communicated by the public methods*** (and features☹) it exposes. More on this when we look at LSP.

Single Responsibility Principle

- *SRP does not limit what a class can do!*
  - The single responsibility could be a very ***large one***. E.g., the grid in the game of Battleships.
  - Often results in less, but fatter (and not God!), classes as responsibility is not distributed throughout an application.
- *Each responsibility should be a separate class, because each responsibility is an axis of change.*

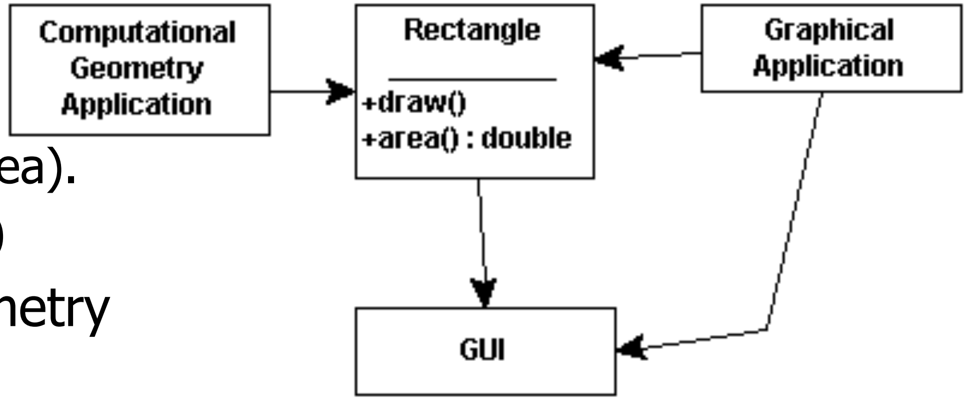
```
interface Modem{
  public void dial(String pno);
  public void hangup();
  public void send(char c);
  public char recv();
}
```

- Modem interface looks reasonable - but ***there are two responsibilities shown***:
  - Connection management
  - Data communication

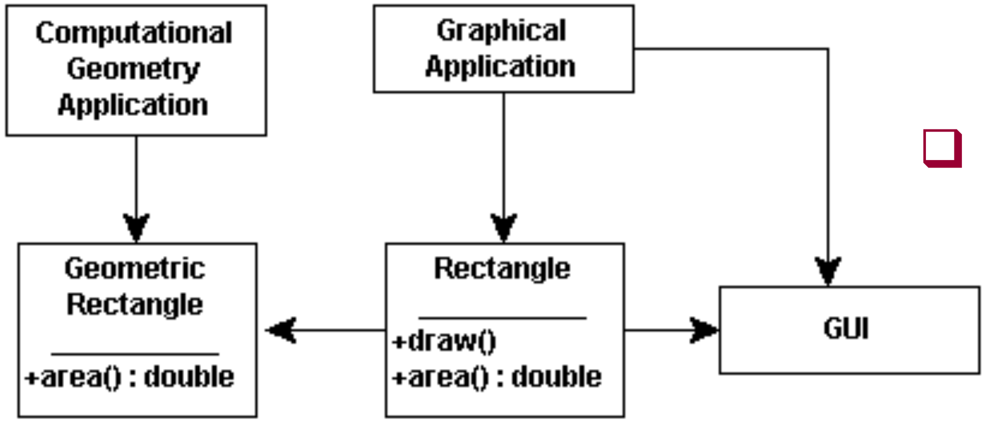
The two sets of functions *have nothing in common*. Will change for different reasons. Called from different parts of app that uses them.

Single Responsibility Principle

- Rectangle class has *two responsibilities*:
  - Mathematical model (area).
  - Render on a GUI (draw)
- Must include GUI in geometry application.
  - Distribute class files.



- Change in Graphical Application requires us to recompile, test & redeploy Computational Geometry Application.
- *Solution is to separate the responsibilities into different classes.*



Liskov Substitution Principle

- *If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2 then S is a subtype of T.*
  - *Barbara Liskov, Data Abstraction and Hierarchy, 1988.*
- *Methods that use references to base types must be able to use references to derived types without knowing the difference.*
  - Importance of LSP is best illustrated in its violation:
  - ***A class referencing a non-LSP conforming class must know about all the derivatives of that base class.***
  - LinkedList class (handout) is a violation of LSP.
- *A model viewed in isolation cannot be meaningfully validated.*
  - ***Must be viewed in terms of reasonable assumptions that will be made by users of that design.***
  - Consider a base class ***Rectangle*** with *setWidth()* and *setHeight()*. What happens when the derived class is a ***Square***? Problems...
  - The ***is-a rule*** does not appear to hold in this case...

Liskov Substitution Principle

- The *is-a* relationship is not so straightforward:
  - A square ***is-a*** rectangle. But a Square object is definately NOT a rectangle object because ***behaviourly they are different***.
- ***LSP makes it clear that in OOP the is-a relationship pertains to behaviour .***
  - ***Not intrinsic private behaviour, but extrinsic public behaviour that clients depend upon.***
  - The author of a Rectangle class depended on the fact that the height and width are independent of each other. ***Independence of two variables is an extrinsic public behaviour***.
- If a program conforms to the OCP, it will automatically conform to the Liskov Substitution Principle.
  - All derivatives must conform to the behaviour clients ***expect*** from the base classes that they use.
- Strong relationship between LSP and Design by Contract.

Design by Contract (DBC)

- *Based on conceptual metaphor of a business contract.*
  - Interface specifications should be ***defined precisely*** (ADTs), and be ***verifiable***.
  - From Eiffle [Bertrand Meyer, 1988]. Aka Programming by Contract, Contract First Programming.
- Precision achieved using *assertions* and *method signatures*.
  - Verification and proofs from formal methods, e.g. Z/Object-Z.
- Like a business contract, there are at least two-parties.
  - Contract imposes obligations/responsibilities in return for benefits.
- Methods and classes declare *preconditions* and *postconditions*.
  - ***Obligations:*** Preconditions must be true in order for method to execute. Method free from cases outside precondition.
  - ***Benefit:*** Post-condition, guarantee by a method of what it will do if preconditions are met.
- Can be implemented using DBC tools like *jContract*.
  - Can also use Java *assert* statement to achieve this.

Design by Contract (DBC)

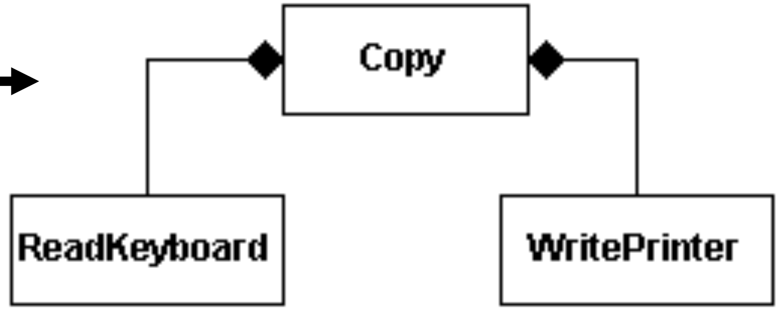
- *Contract for a method contains the following and their meaning.*
  – Acceptable/unacceptable ***input types***. Return values or types.
  – ***Exception*** conditions and types.
  – ***Preconditions***, which subclasses may weaken (but not strengthen).
  – ***Postconditions***, which subclasses may strengthen (but not weaken).
  – ***Predicates***, invariants that always retain their truth value.
- *When redefining a routine in a derivative, you may only replace its precondition by a weaker one, and its postcondition by a stronger one. - Bertrand Meyer, 1988.*
- ***Define a contract on each method.***
  – ***Precondition:*** constraint on object state before a method is called. Eg, precondition to calling *Iterator.next()* is that *hasMore()* is true.
  – ***Postcondition:*** constraint on object state after a method call has completed. E.g. a *List* is not empty after calling *add()*.

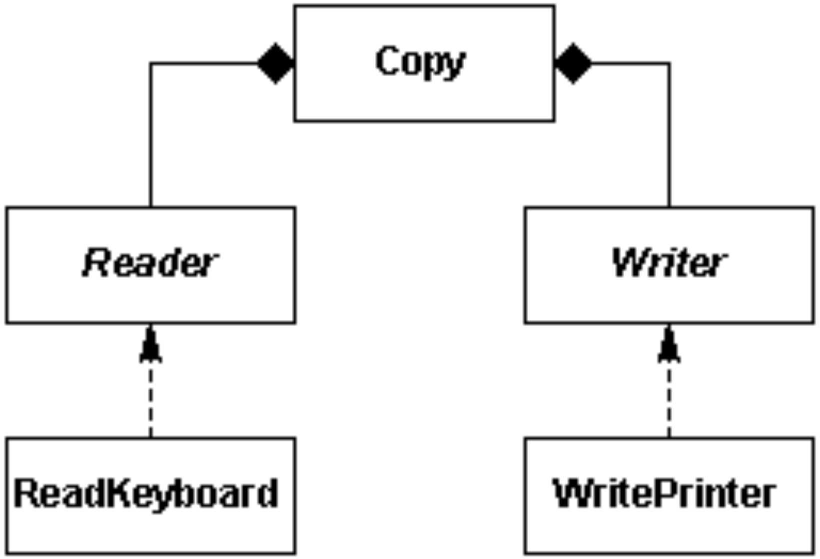
Dependency Inversion Principle

- *A) A high-level module should not depend on low-level modules. Both should depend upon abstractions.*
- *B) Abstractions should not depend upon details. Details should depend upon abstractions.*
- Violation of this principle leads to rigid, fragile and immobile code:
  - Traditional structured analysis and design methods give rise to higher level modules that are ***highly coupled*** with low level subroutine libraries.
  - Changes to low level classes require changes to the high level modules we want to reuse. ***DIP inverts this dependency***.
- DIP is at the heart of framework design.
  - ***Separate interface from implementation.***
- *Lower level modules defined in terms of abstract classes. Higher level module only interact with abstract interfaces.*
  - Need to ensure that there are no ***transitive*** dependencies.

Dependency Inversion Principle

- *Copy* copies characters typed on a keyboard to a printer.
  - Copy contains a loop that calls *ReadKeyboard* to read from keyboard and send character to *WritePrinter*.



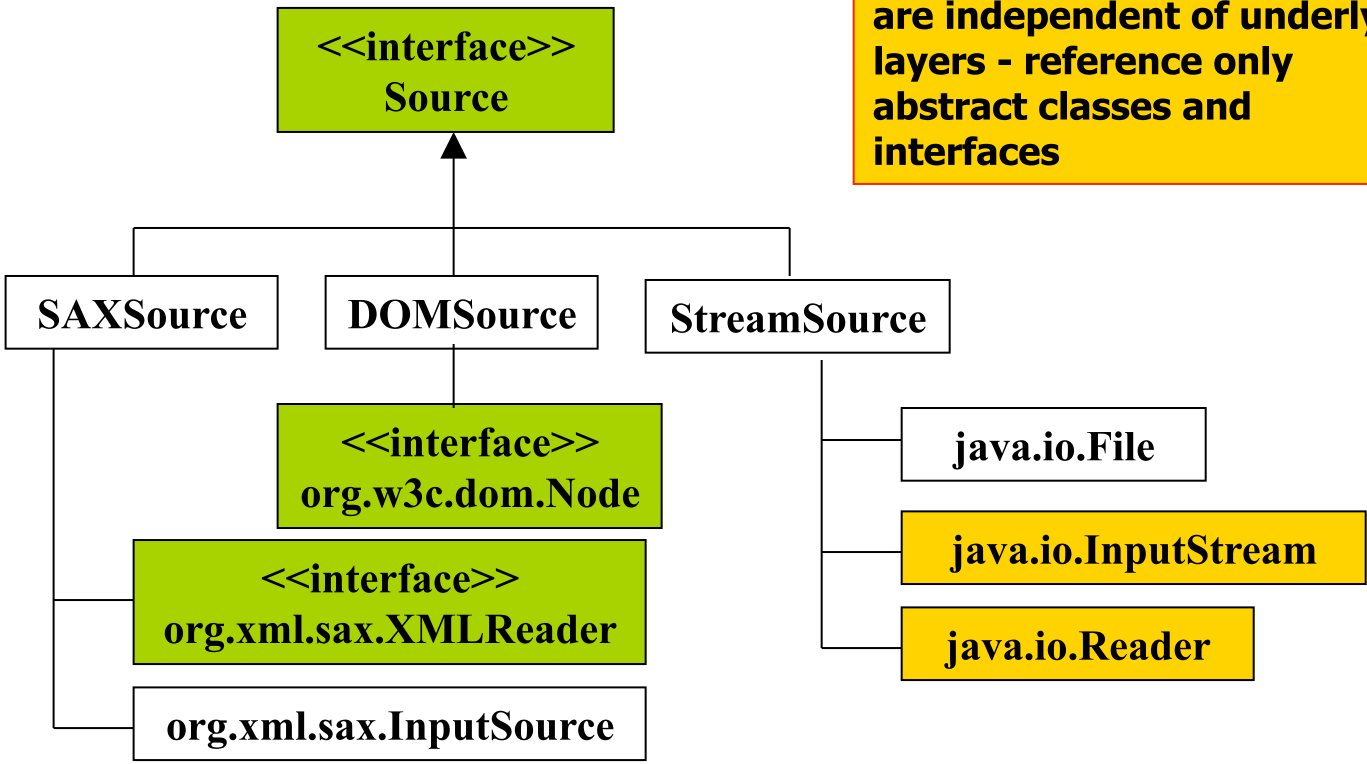


- Copy has a dependency on keyboard and printer modules.
  - ***<u>A high level "policy" depends upon its details.</u>***
  - *Reader* and *Writer* abstract classes ***<u>invert the dependency.</u>***
  - Can now use *Copy* with many types of IO devices ***provided they are subtypes of Reader and Writer***.
  - Copy exemplifies OCP.

Dependency Inversion Principle

- Consider the design of JAXP I/O. Allows input sources to be specified in terms of a file name, URL, byte/char stream, DOM and SAX source...flexible.

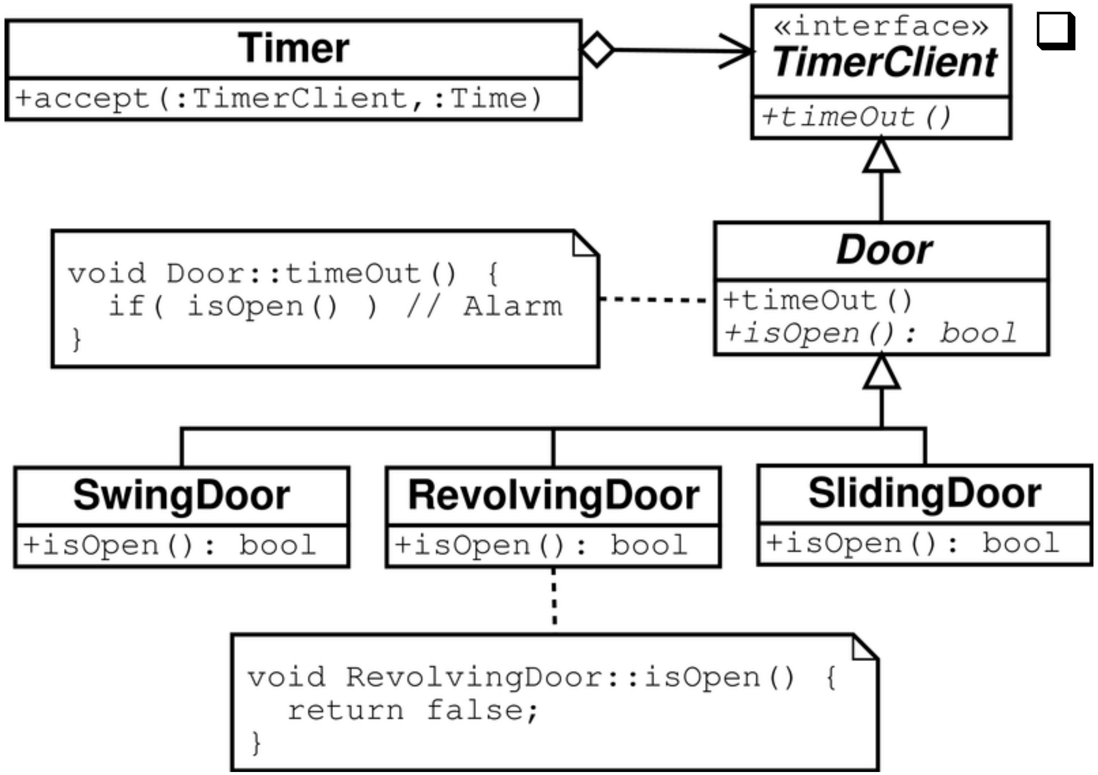
Higher level Source modules are independent of underlying layers - reference only abstract classes and interfaces



Interface Segregation Principle

- *Clients should not be forced to depend on interfaces that they do not use.*
- Client should *not know about method groups* as a single class.
  - ***Should know about abstract base classes***, one for each method group, which have cohesive interfaces.
- Classes with ***fat interfaces*** have interfaces that are not cohesive.
  - Methods can be broken into groups of functions that serve different client needs. Multipurpose classes that are not fully abstracted.
- ***Inadvertent coupling between the clients that share single class interfaces.***
  - If additional functionality added to interface to support a client, then ***all other clients will be affected*. Propagates change.
- *Many client specific interfaces are better* than one general purpose (God) interface.

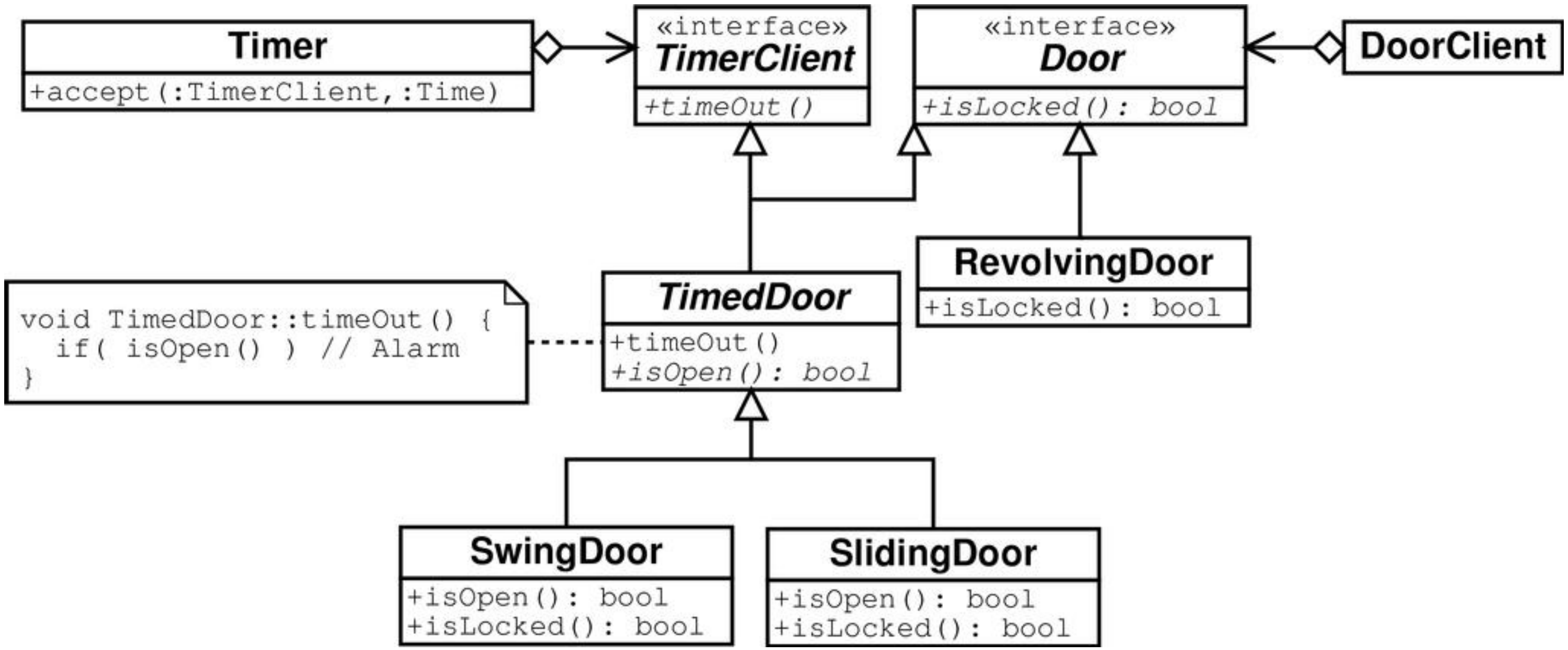
Interface Segregation Principle



- *Door* is abstract class.
  - Clients can use objects that conform to the *Door* interface, without having to depend on implementation of *Door*.
  - A timed door sounds an alarm when door is left open too long.

- May be derived classes of *Door* which don't need the *TimerClient* interface.
  - They suffer from depending on it anyway.

Interface Segregation Principle



- *RevolvingDoor* does not depend needlessly on *TimerClient*.
  - *SwingDoor* and *SlidingDoor* really are timed doors.
- <u>*Multiple inheritance*</u> solves the fat interface problem for us.

Granularity of OO Software

- As software applications grow in size, some form of high-level organisation is required.
  - A class is a convenient unit for organising small applications.
  - ***Too fine grained*** for large applications. Larger granule required.
- ***Packages*** (aka Class Category [Booch], Clusters [Meyer], Subject Areas [Coad]) *represent a logical grouping of declarations that can be imported into other programmes*.
  - Enables us to reason about design at a *higher level of abstraction*.
  - We partition classes in an application using some criteria. Allocate partitions to packages.
- ***Raises a number of questions:***
  - What are the best ***partitioning criteria***? What degree of *cohesion*?
  - What are the ***relationships*** that exist between packages and what are the design principles that govern their use?
  - Should packages be designed before classes (***top-down***) or vice-versa.
  - How are packages ***physically represented***?

Reuse-Release Equivalence

- *The granule of reuse is the granule of release. Only components that are released through a tracking system can be effectively reused. This granule is the package.*
- Code copying is *not a form of reuse*.
  - Changes to original and copy result in major problems when faced with an update from original author.
- Reuse refers to when you *do not have to look up source code.*
  - Just lookup static/dynamic library. Whenever these libraries change, you can import a new version when opportunity allows.
- Reusuable code treated like a product. Not necessarily maintained/ distributed by client.
  - However, clients ***must be notified*** when changes are made.
  - Author should demarcate releases with numbers/names.
- ***Can reuse nothing that is not also released.***
  - Users of a released library are ***clients of the entire library.***
- *Packages are natural candidates for a releasable entity* (granule).

Common Reuse Principle (CRP)

- *The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.*
- Provides guidance on which classes should be used in a package.
  - Classes that tend to be reused together *belong in the same package*.
- *Reusuable classes normally collaborate with other classes that are part of a reusuable abstraction.* Should be packaged together.
  - When we use a package, ***a dependency is created*** upon the whole package.
  - Everytime a package is released, applications using it ***must be revalidated and re-released***. If we are only using a small portion of a package, this can be time consuming and wasteful.
- Packages commonly manifested physically as shared libraries (JAR archives) and DLLs.
  - If a DLL is released because of a change to a class that I don't use, I still must redistribute the new DLL and revalidate the application.

Common Closure Principle (CCP)

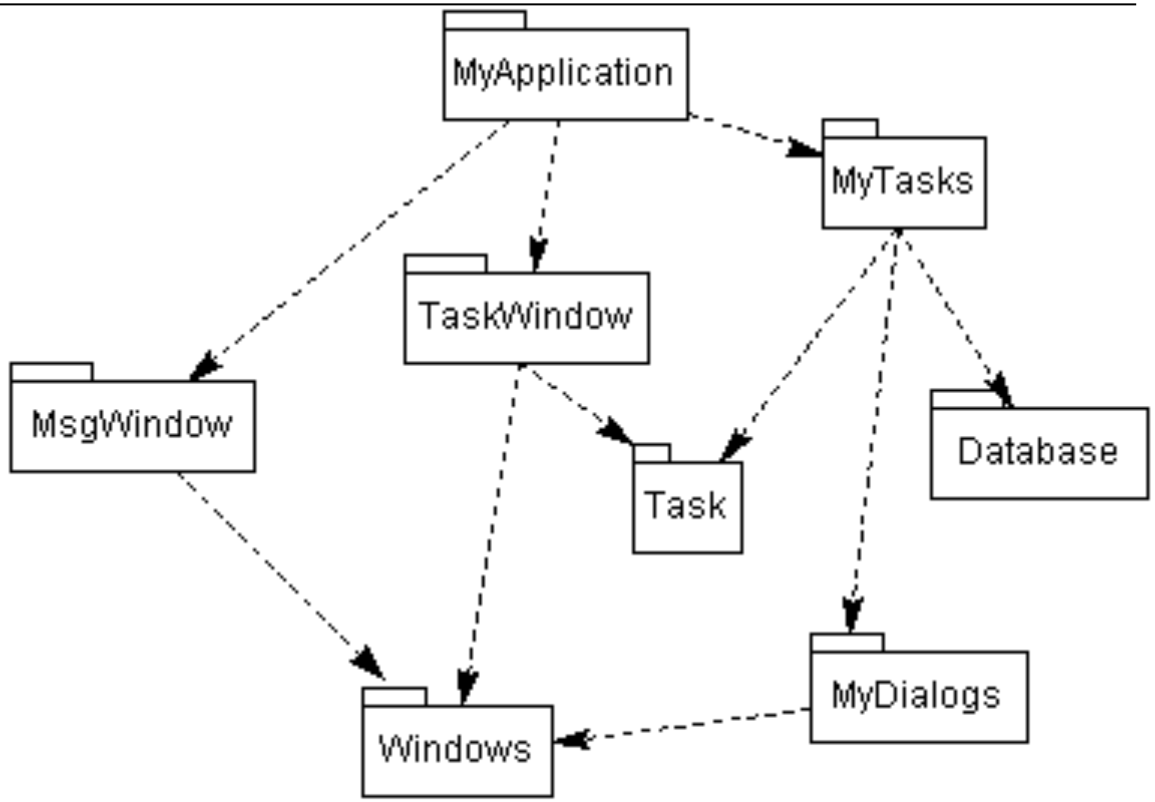
- *The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package.*
- Maintainability is *more important* than reusability.
- CCP attempts to *gather in one package, all classes that are likely to change for the same reasons*.
  - ***If two classes are so tightly bound, either physically or conceptually, that they almost always change together, then they belong in the same package.***
  - Minimises workload relating to releasing, revalidating and redistributing.
- *Closely associated with the Open-Closed Principle.*
  - CCP packages together classes that cannot be closed against certain types of change.
  - When a change in requirements arises, the change can be restricted to a minimal number of packages.

Acyclic Dependencies Principle

- *The dependency structure between packages must be a Directed Acyclic Graph (DAG). That is, there must be no cycles in the dependency structure.*
- "Morning after syndrome" - many developers modifying same source file.
  - One developer makes changes (après pints...) and causes chaos.
- *Solution is to partition development logic into releasable packages.*
  - Packages become ***a unit of work*** that are the responsibility of a development team.
  - Package given a ***release number*** (stable build) and released for use by other teams.
  - Team continue to modify package in ***isolation***. Other teams use ***released version only***.
- None of the teams are at the mercy of the others.
  - Only works if we manage the dependency structure of packages. ***There must be no cycles in the dependency structure.***

Acyclic Dependencies Principle

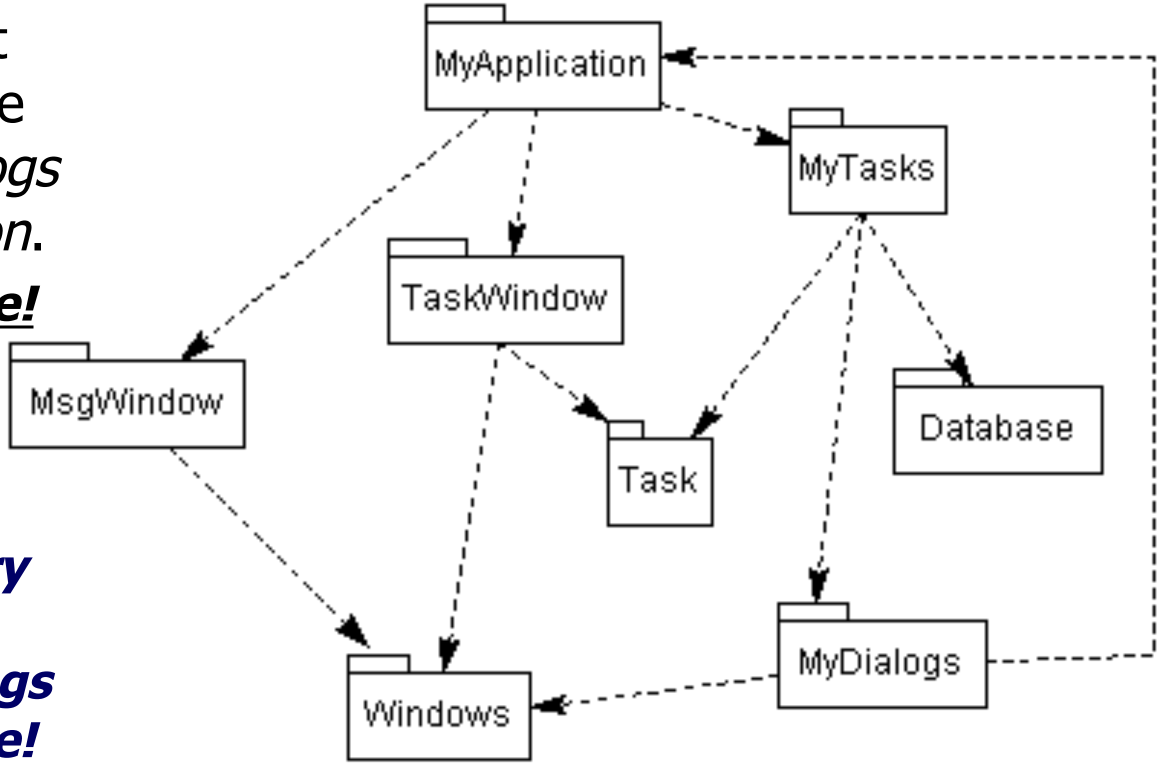
- *DAG* - packages are nodes. Dependencies are edges and are directed.
- ***NO cycles!***
- Changes to *MyDialogs* will only affect *MyTasks* and *MyApplication*.
- ***No affect on the other packages!***



- Unit ***test*** of *MyDialogs* involves recompile and test with *Windows*.
  - System release done from ***bottom up***: *MsgWindow, MyDialogs, Task, TaskWindow, Database and MyApplication* respectively.

Acyclic Dependencies Principle

- New requirement forces a reference between *MyDialogs* and *MyApplication*.
- ***Now have a cycle!***



- ***MyTasks now depends on every package in the system. MyDialogs suffers same fate!***

❑ *Cyclic dependencies* make release of these classes very difficult.
  – Have in effect become one large package.
  – ***Unit test of MyDialogs will now require a complete system build.***

Acyclic Dependencies Principle

- There are *ways to beak the cycle* and reinstate DAG:
  - ***Apply Dependency Inversion Principle:*** create an abstract class with interface that MyDialogs needs. MyApplication subclasses the abstract type. Inverts the dependency and breaks the cycle.
  - ***Create a new package*** that both MyDialogs and MyApplication depend on. Migrate class(es) they depend on to new package.
- Latter option implies package structure not stable ("jitters").
  - As application grows, dependency structure & jitters grow also...
- ***Package structure cannot be designed from top-down.***
  - Do not start with packages and then create classes.
  - Package dependency diagrams are map of ***how to build a system***.
- Use *Common Closure Principle* and co-locate classes that are likely to change.
  - As application grows, reusuability facilitated by *Common Reuse Principle*.

Stable Dependencies Principle

- *The dependencies between packages in a design should be in the direction of the stability of the packages. A package should only depend upon packages that are more stable than it is.*
- Stability is not a measure of the likelyhood that a module may change. It is a measure of the difficulty in changing a module.
  - The more difficult to change, ***the less volatile*** the module.
- Designs cannot be static. Some volatility required to maintain the design. Accomplished using *Common Closure Principle*.
  - Create packages that are sensitive to certain kinds of changes. These packages are ***designed*** to be volatile. We ***expect*** them to change.
- *Any package that is expected to be volatile should not be depended upon by a package that is difficult to change.*
  - Otherwise the volatile package will also be difficult to change.

Stable Dependencies Principle

- Stability of a package can be measured by counting the number of dependencies that enter and leave a package.

  Positional Stability (I) = (Ce / (Ca + Ce))

  - ***Ca: Afferent Couplings:*** number of packages outside this package that depend upon classes in this package.
  - ***Ce: Efferent Couplings:*** number of classes inside this package that depend upon classes outside this package.
- *Metric has the range [0,1].*
  - 0 implies a *maximally stable package*: depended on by many packages. Does not itself depend on any others. ***Responsible and Independent***. Difficult to change package.
  - 1 implies *maximally instable package*: ***Irresponsible and Dependent***. Lack of dependents give it *no reason not to change* - packages it depends may give it ample reason *to change*.
- Metrics should decrease in the direction of dependency.
  - Dependencies correspond to ***import/include*** statements.

Stable Abstractions Principle

- *Packages that are maximally stable should be maximally abstract. Instable packages should be concrete. The abstraction of a package should be in proportion to its stability.*
- *Describes the relationship between stability and abstraction.*
  - ***A stable package should also be abstract*** so that its stability does not prevent it from being extended.
  - ***Instable packages should be concrete*** since instability allows concrete code within to be changed.
- Reader/Writer (Copy example) are abstract and highly stable.
  - Depend upon nothing and are depended upon by copy and all derivatives. But still can be extended to deal with different IO.
- *SAP and SDP combined amount to the Dependency Inversion Principle for packages* (although not as clear-cut)
  - SDP implies that dependencies *should run in direction of stability*.
  - SAP states that *stability implies abstraction*.