(11224) INTRODUCTION TO SOFTWARE ENGINEERING

Lecture 9: Design Issues

Shereen Fouad

Announcement

- You need to sign yourself up to your chosen group (latest) by today (the 9th of February 2015) no later than 5pm. After this date/time I will have to sign you up to individual groups.
- No afternoon lecture (3pm-4pm) next Monday the 16th of Feb. It will only be one morning lecture (11am-12pm)

Tame & wicked Problems

- **Systems** are difficult to work with, and seeing things for what they are is an essential first step.
- Horst Rittel in the late 1960s distinguished between ***"tame"*** and ***"wicked"*** problems.
- This is not the distinction between easy and hard problems!!

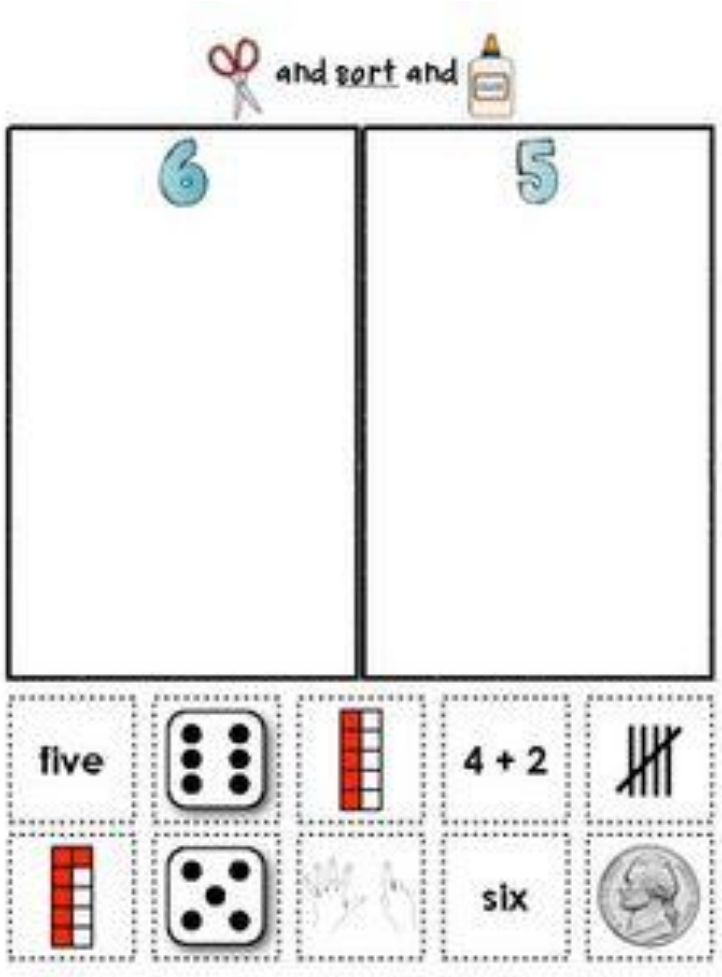
Tame problems characteristics

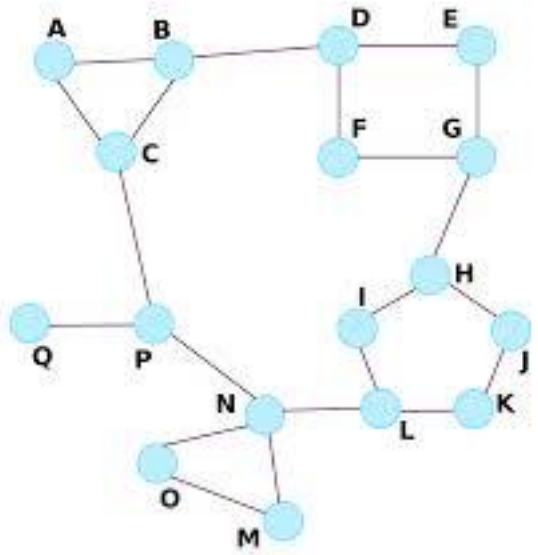
- Clear, well defined descriptions which do not change.
- Very easy to see when you have finished solving the problem.
- Proposed solution is either right or wrong, there are no halfway measures.
- The problem is one of a number of similar problems which can all be solved in similar ways.
- Proposed solutions are easy to check and throw away if they do not work.
- They have a limited number of alternative solutions.

Writing a program to solve Sudoku puzzles

- Sudoku puzzles are very definite, clear problems that are easy to understand, though it is not necessarily easy to solve the problem.
- Designing a program to solve such puzzles is also a problem which is clear and easy to understand, even if the actual design of such a program is not itself easy.
- It is relatively easy to see when you have finished the design of such a program.
- Once we have a design, it is either right or wrong: it is right if it can solve any possible valid Sudoku problem, otherwise it is wrong.

Examples of Tame problems

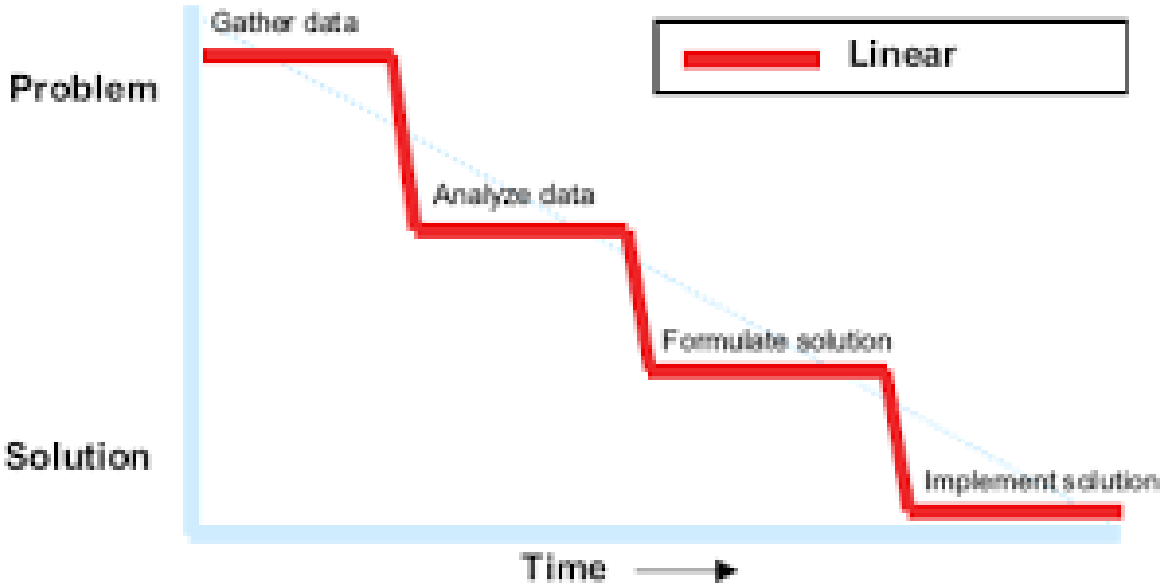




More About Tame Problems

- Tame design problems are not necessarily easy. Sometimes they are very hard indeed.
- They are normally solved by directly using knowledge of standard computer science algorithms and data structures.
- Since tame problems have clear descriptions, we could imagine that one day we might be able to write those descriptions formally, and have a program that takes the formal description and generates a working program that solves the problem. In fact, for certain simple and special cases, that is indeed already possible.

Tame problems can often be solved in a fairly linear fashion

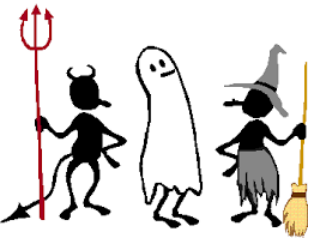


Wicked problems characteristics

- Generally the problem is not understood until a solution has been built.
- There is typically no clear point at which to stop work on the problem
- Solutions to wicked problems do not have an absolute, black or white, right or wrong nature. Instead they have a relative nature.
- A wicked problem is novel and unique. One typically cannot adapt a solution to one wicked problem to implement a solution to another one

Wicked problems characteristics

- Evaluating proposed solutions is hard, and may involve subjective judgements.
- Wicked problems have no standard set of alternative solutions from which one can pick.
- Most of the programs that professional programmers are called on to design and implement, and most of the final year projects in your degree programme, are of this wicked class.



Examples of Wicked Problems

1. Translate a Chinese text file into English
2. Build an automatic air traffic control system.
3. Build a web application to handle auctions of items submitted by users.
4. Build an online application to allow multiple users to collaborate interactively in developing a UML diagram.
5. Build a text editor for editing programs.
6. Build an OCR program to turn scanned images of the pages of a book into the editable source for the formatted book.

What are the properties of Good Designs?

Properties of Good Designs

Correctness

- The most important property is that the design must work correctly
- Reliability is a sub-issue of correctness

Properties of Good Designs

Simplicity

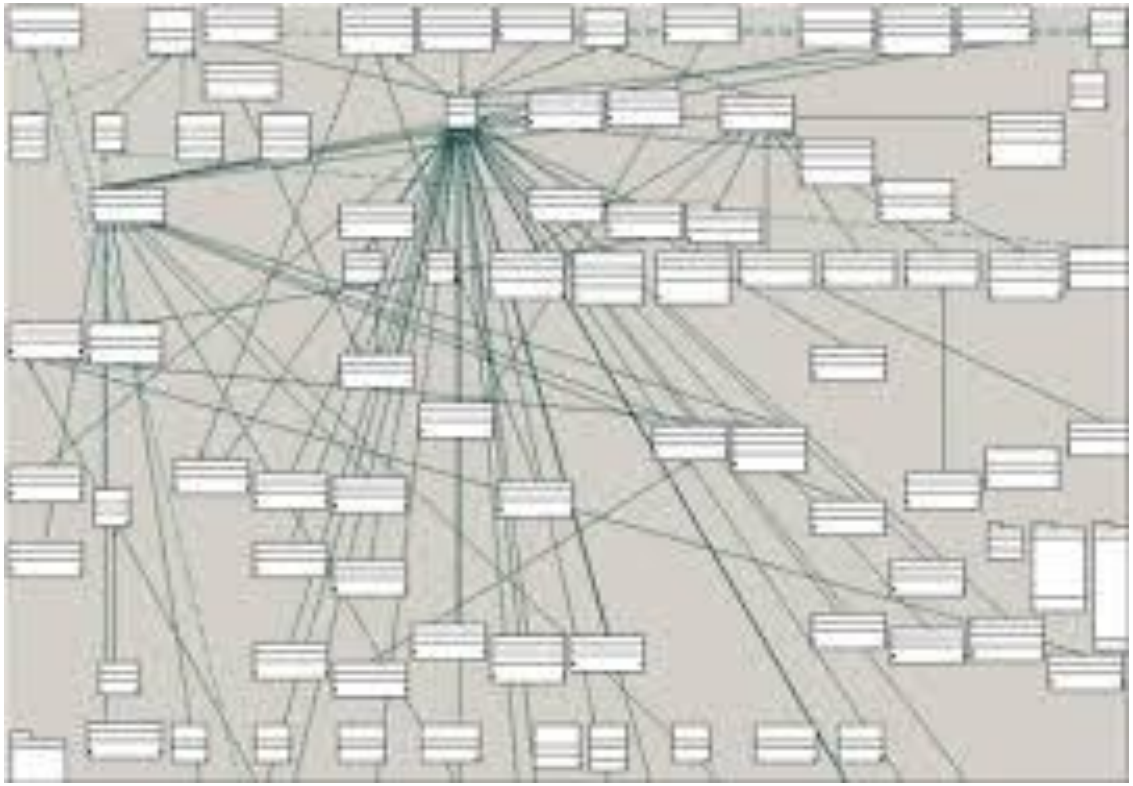
- The more complex the design, the harder it is to get the design right and to implement it.
- An unnecessarily complex design leads to systems that are harder to maintain as well, because more factors and details have to taken into account and checked when any change is required.

What is the relationship type in UML class diagram that you may ignore in order to achieve simplicity?

According to Neeraj Sangal is president of Lattix Inc. company

“Excessive inter-module dependencies have long been recognized as an indicator of poor software design. Highly coupled systems, in which modules have unnecessary dependencies, are hard to work with because modules cannot be understood easily in isolation, and changes or extensions to functionality cannot be easily localized.”

http://www.eclipsezone.com/articles/lattix-dsm/?source=archives



Properties of Good Designs

Maintainability

- Software systems require change. **Why??**

Software changes arise for a number of reasons

- errors in early versions of the design or implementation,
- requirements which are only discovered or understood late in the design process,
- requirements that have changed since the project started because of changes in the business or new opportunities that have arisen, etc.

If the design is simple, clear, well documented and matches the implementation well, making such changes is faster, easier, less costly and less likely to introduce further problems.

Properties of Good Designs

Usability

- A system that does the right thing all the time, but is so difficult, annoying or inconvenient to use that nobody actually uses it is not a satisfactory solution to the problem.

What is usability?

- Usability can simply be thought of as the practical implementation of good software, but, more formally :
- A usable system is:
  - easy to use
  - easy to learn
  - easy to remember how to use
  - effective to use
  - efficient to use
  - safe to use
  - enjoyable to use

Game !!

- What's wrong with each design?
  - Type of error
  - Who is affected
  - Impact
- What's a redesign solution?

Good or Bad Design?

https://kanrawi.wordpress.com/category/goodbad-interface-design/

Good or Bad Design?

GOOD OR BAD DESIGN?

GOOD OR BAD DESIGN?

https://kanrawi.wordpress.com/2012/10/13/bad-design-milk-container/

https://kanrawi.wordpress.com/2012/10/13/bad-design-milk-container/

https://kanrawi.wordpress.com/2012/10/13/bad-design-milk-container/

HOW TO IMPROVE THE BELOW DESIGN

https://kanrawi.wordpress.com/2012/10/13/bad-design-milk-container/

Portability

Portability

- Requirements are not the only parts of the environment of software systems that change over time.
- Changes to the hardware that these systems must run on, new operating systems on existing hardware have to be handled, new libraries for building interfaces arrive and must be exploited.
- Therefore a good design should expect such changes and minimise their impact, in spite of the sometimes unpredictable nature of such changes.
- Portability is the key issue for development cost reduction.

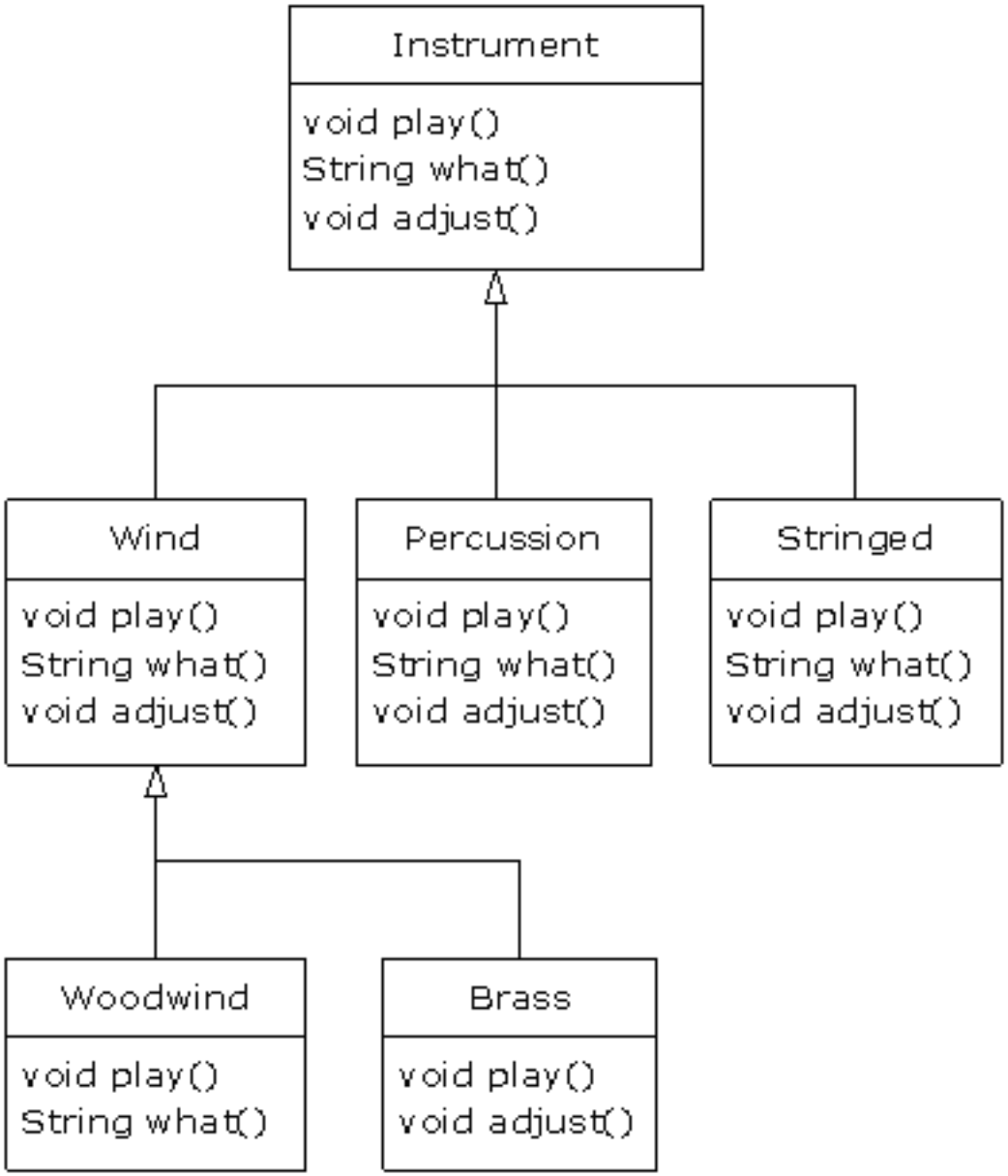
Description of a Portable Java game

- The game will be written in Java, this makes it able to run on lots of different devices.
- We will be writing out game for use with PCs, it will be able to be played on any PC running JRE with a keyboard and mouse.
- It can work on different operating systems, including windows, Linux and Unix

Extensibility

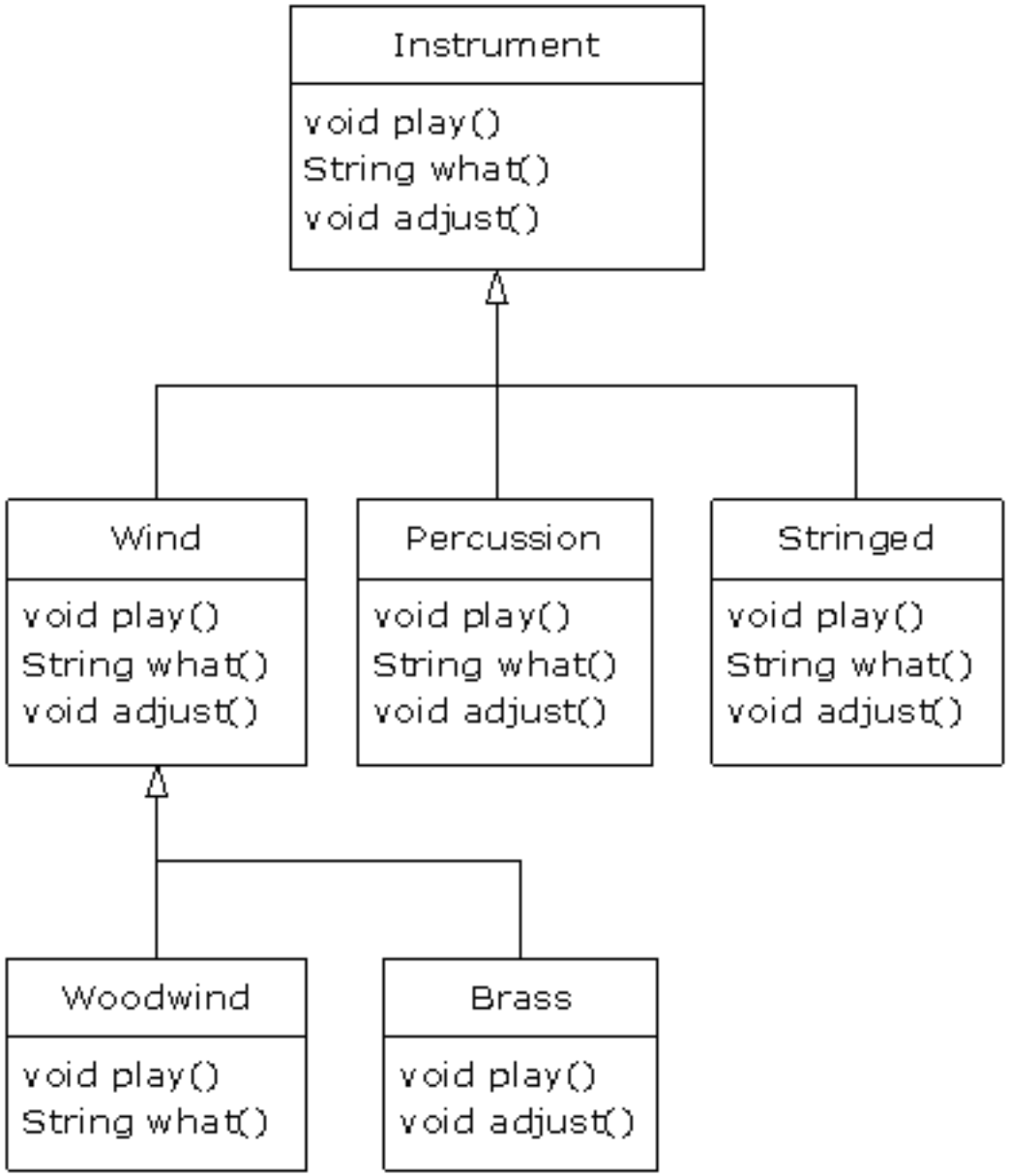
Extensibility

- Goes beyond maintainability
- over the lifetime of a software system, we can expect new functionality to be required that was never part of the original plans for the system.
- new requirements that we can not plan specifically for them at the start of the project,
- prepare our design in such a way that whatever new functionality is required, the effects on the system design and implementation as a whole is minimised.



What if I want to add new functionality to all of the instruments which is tune()?

Is this an *extensible* design?



- Such a program is *extensible*
- because you can add new functionality by inheriting new data types from the common base class.

Good Design Heuristics

Abstraction

- Programs involve management of huge numbers of details,
- Example:
- In software systems: we may have to decide that we are going to implement a particular collection of objects using a doubly linked list, but when using that collection we need to abstract away from such details, clearing our head of them, and just remember that we have a collection of objects.

Why??

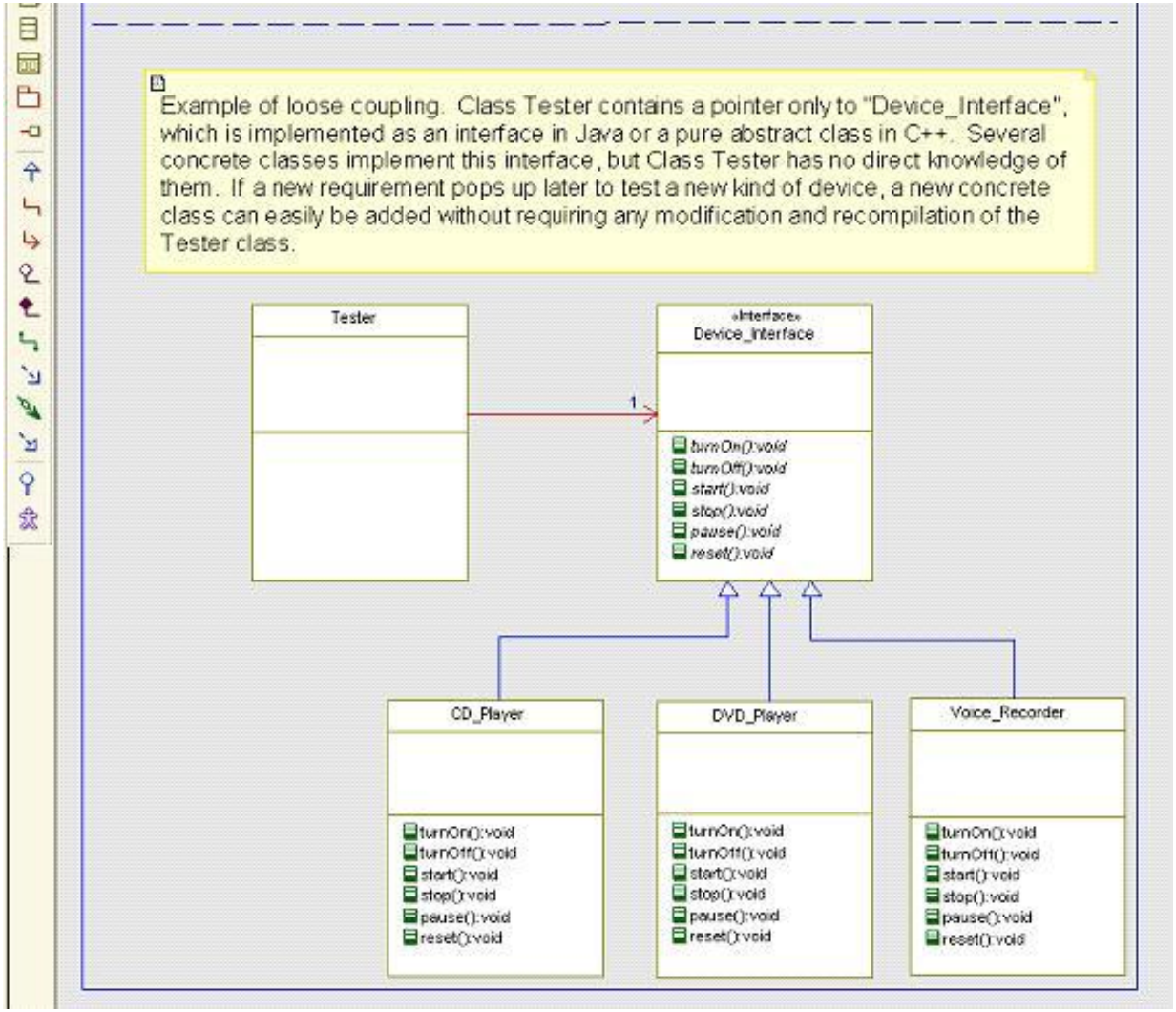
- Ensure that your designs allow you to hide or defer consideration of details, thus reducing complexity
  - A good abstraction is said to provide *information hiding*
  - Abstractions allow you to understand the essence of a subsystem without having to know unnecessary details

Good Design Heuristics

Loose Coupling

- Coupling is a measure of how deeply different modules depend on internal knowledge of each other.

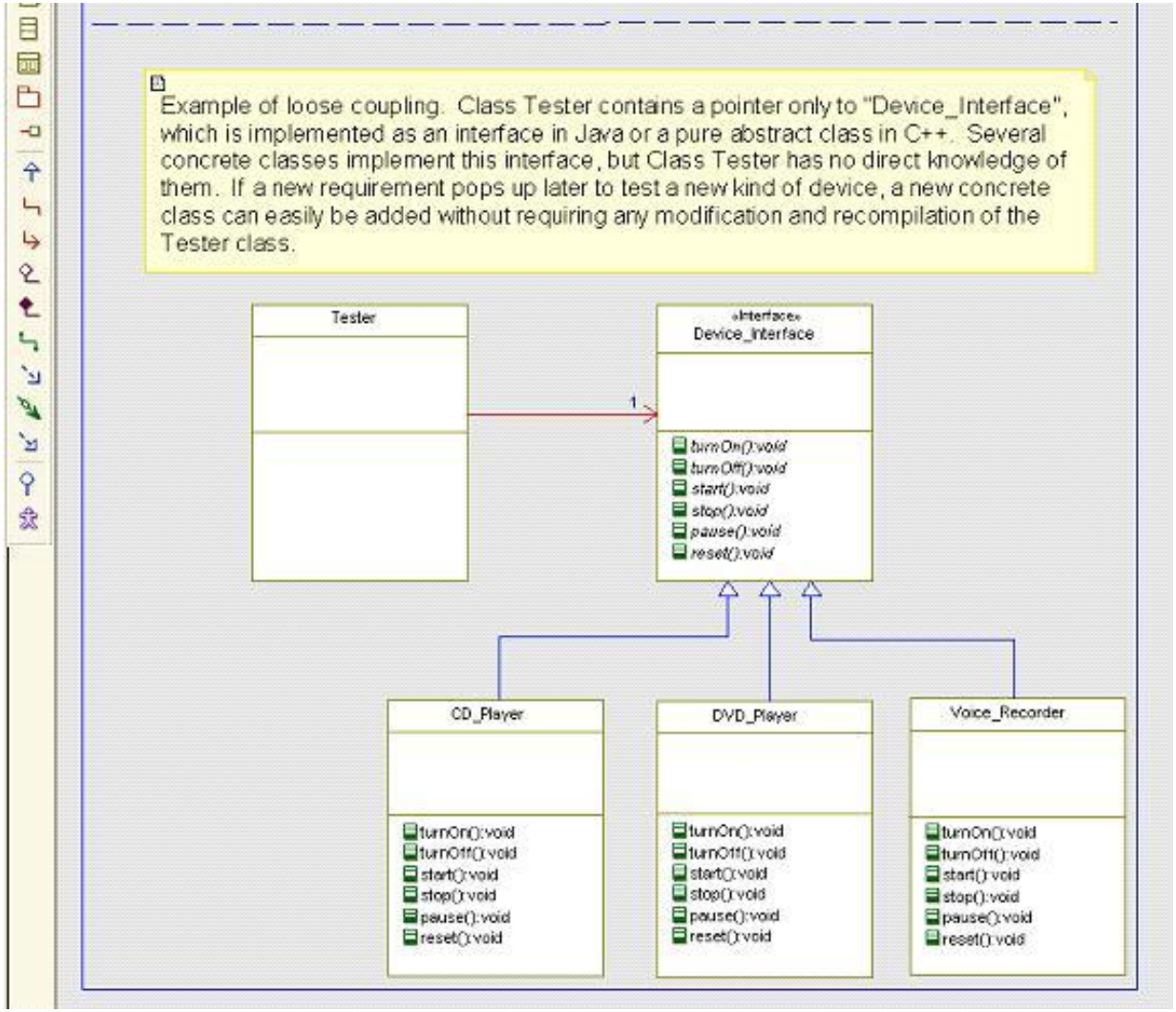
If one module uses another module only through its published interface, and that interface exposes no internal details of how the module is, or might be, implemented, then the coupling between them is loose.



http://en.wikipedia.org/wiki/Loose_coupling

Why?

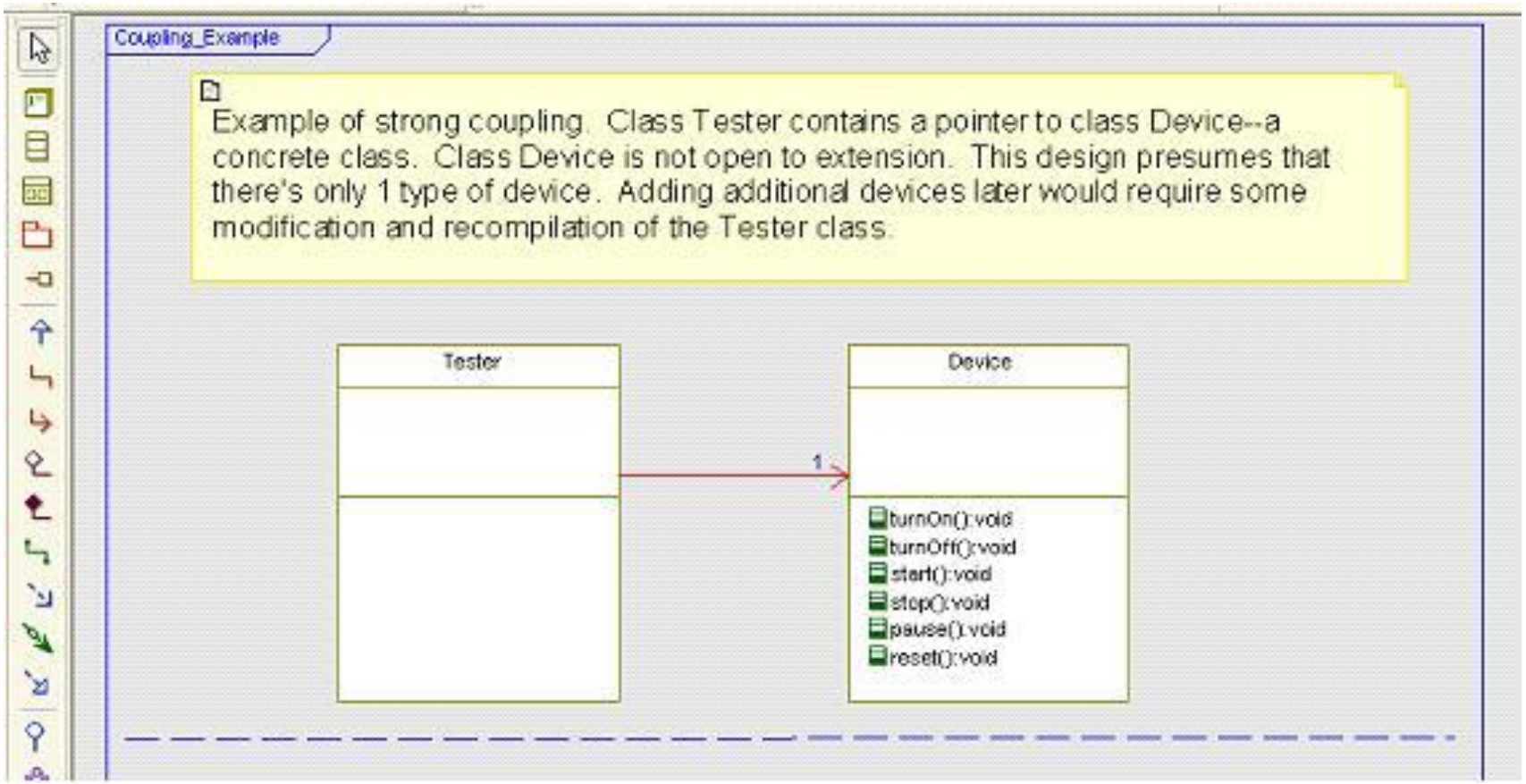
If one module uses another module only through its published interface, and that interface exposes no internal details of how the module is, or might be, implemented, then the coupling between them is loose.



http://en.wikipedia.org/wiki/Loose_coupling

Loose coupling allows modules to completely change their internal design or implementation without requiring any changes to other modules that use

tight or strong coupling means that changes to one module's design or implementation will typically require significant changes to other modules.



http://en.wikipedia.org/wiki/Loose_coupling

Modularity

- Modules are a way of directly encoding separation of concerns
- Modules are pieces of code that implement one concern and that present an interface to the rest of the system through which all external parts of the system must communicate in order to interact with the module.
- Modular programming is based around concepts of strict information hiding, encapsulation and interface control.
- If a programming language does not explicitly support modules, then programmers can still program in an modular fashion in those languages by keeping to appropriate conventions.

i. Fortran subroutine

ii. Ada package

iii. Procedures & functions of PASCAL & C

iv. C++ / Java classes

v. Java packages

vi. Work assignment for an individual programmer

Good Design Heuristics

High Cohesion

- This is to do with how focused each individual module is.
- While there are different types of cohesion, the best type is functional cohesion.
- In a system with high functional cohesion, each module collects together the code to handle a single functional goal.
- In a system with low functional cohesion, the classes, operations and methods may be unrelated by function,

Why it is important?

Why it is important?

- If the code for a specific piece of functionality must be updated or changed, multiple different modules must be modified and the work is hampered by the problems of separating out the parts of each module that must be worked on from all the other parts in the module that are not relevant.
- If the system has high functional cohesion, such a change will be limited to a single module and the code is easier to read and understand as it is all related to the same functionality.

This might represent **higher cohesion**

```
class MyReader{
 public MyData readFromDisk(Parameter fileName){...};
 public MyData readFromWeb(Parameter url){...};
 public MyData readFromNetwork(Parameter networkLoc){...};
}
```

Look how well-focused the class is in its purpose. The class is named "MyReader" and its intended purpose is to read the resource, and it does only so. It does not implement other things. Its highly cohesive. Lets look at low cohesion example.

```
/*
 * lower cohesion example (probably)
 */
class MyReader{
 //validate the resource path
 public boolean validateLocation(String path){
  return ping(pathIP) && checkFTP(path);
 }
 private static boolean ping(String path){...};
 private static boolean checkFTP(String path){...};

 //read the resource
 public MyData readFromDisk(String fileName){...};
 public MyData readFromWeb(String url){...};
 public MyData readFromNetwork(String networkAddress){...};
}
```

Well, the read operations are well defined, but it also implements logic to validate path. This is just an example and describes lower cohesion, wherein, a class intended for a specific functionality is also involved in other operations.

http://javabambino.blogspot.co.uk/2013/06/loose-coupling-and-high-cohesion-in-java.html