SE Software Engineering
The University of Auckland

Quality Assurance
Coding Principles II

Part II - Lecture 13

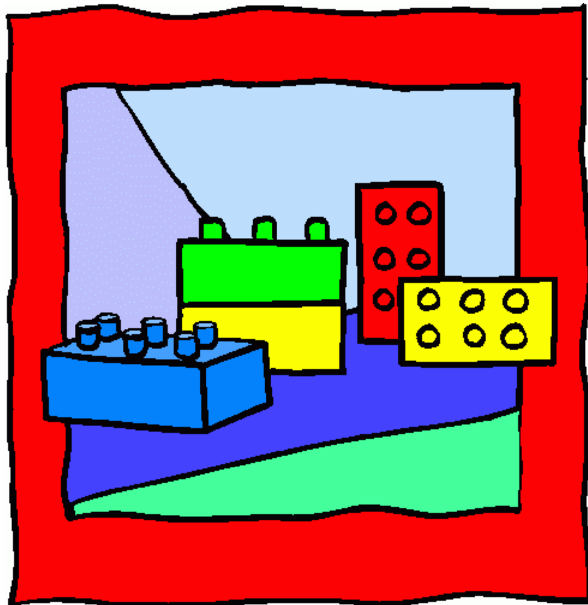
Today's Outline

Software Engineering
The University of Auckland

- Modularity
- Information Hiding

Software Engineering
The University of Auckland

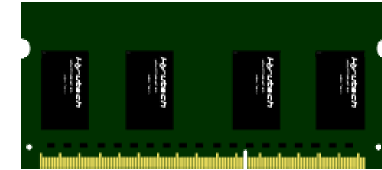
Modularity



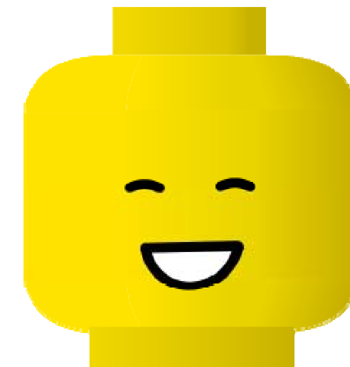
LEGO is not a toy.
It's a way of life.
(Mike Smith)

Modularity

- Complex systems can usually be **divided into simpler pieces** called modules
- **Module:** self-contained component of a system
  - Has a **well-defined interface** to other modules
  - **Separates its interface** from its implementation
  - Usually corresponds to a **set of data types and code**, similar to Java packages



- **Modular systems** (i.e. systems that are composed of modules) are easier to understand, develop and maintain
  - When dealing with a module the details of other modules can be ignored (**separation of concerns**)
  - Modules can be **developed independently**
  - Better **isolation** between modules can prevent failure in one module to cause failure in other modules
  - Modules can be **exchanged** by other modules
  - Modules can be **reused** in several systems



Component Frameworks

- **Component frameworks** support the development and connection of modules
  - Format for specifying module interfaces
  - Functionality for loading, connecting and running modules
- **Module interconnection languages** allow to specify the module interconnections in a system on a high level
  - **Programming-in-the-small**: coding a module
  - **Programming-in-the-large**: assembling a system with modules

Multimedia stream processing

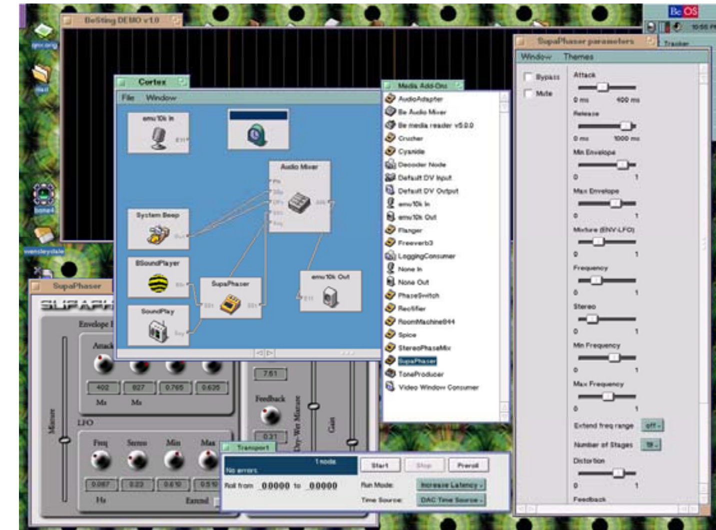
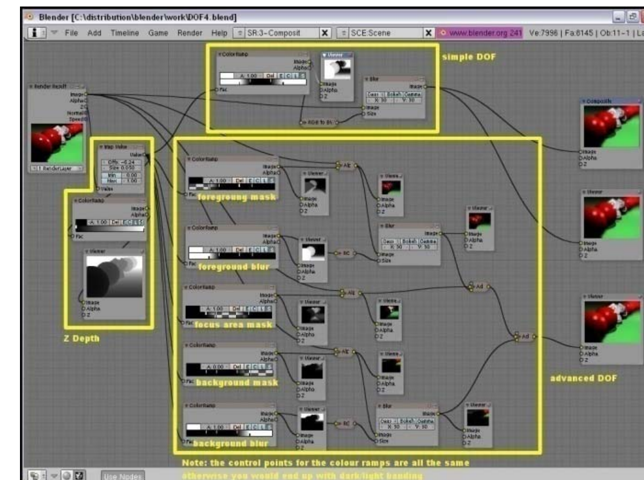


Image processing with Blender



Separation of Concerns

- How to deal with complexity in a system?
- **Separation of concerns** (SoC)
  - Separate issues (break down large problems into pieces) and concentrate on one at a time
  - Break a program into distinct features that overlap in functionality as little as possible
  - **Concern**: a piece of a program, usually a feature or a particular program behavior
- Can be achieved in various ways
  - Traditionally approached through modularity
  - In OO: separate concerns into classes and methods
  - In UIs: separate content from presentation and presentation from application logic
  - Service-Oriented Architecture (SOA): split up functionality into different (web-) services
  - Aspect-Oriented Programming (AOP): separate concerns into "aspects"

Cohesion and Coupling

- Rule for the design of modules: "low coupling, high cohesion"
- A module should be highly cohesive
  - It should form a meaningful, **self-contained unit**
  - The parts in the module **fit and work together** closely
- Between modules there should be low coupling
  - **Little dependencies** between modules
  - A module is independent from the internal implementation of another module
  - Changing the implementation of one module does not require changing other modules
  - The interaction between modules is restricted (through a stable interface)
  - Each module should be understandable without having to understand the details of other modules
- Cohesion and coupling are **usually related**:
  low coupling goes with high cohesion and vice versa
- In OO: less connections between classes (low coupling) if we group related methods of a class together (high cohesion)

Coupling in OO Programming

Software Engineering
The University of Auckland

Coupling is increased between two classes **A** and **B** if:

- **A** has an attribute that has type **B**
- **A** calls a method of **B**
- **A** has a method which uses **B** (return type, parameter or local var)
- **A** is a subclass of (or implements) **B**

```
class A extends B {
  B b1;
  B m1(B b2) {
    B b3 = b1.m2(b2);
    return b3;
  }
}
```

Disadvantages of high coupling include:

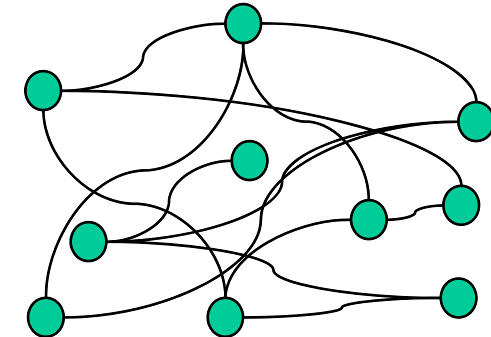
- **Hard to understand** a class in isolation
- Change of one class often **forces changes** in other classes
- **Hard to reuse** or test a class because dependent class must also be available

Software Engineering
The University of Auckland

Spaghetti Code vs. Modular System

Spaghetti Code

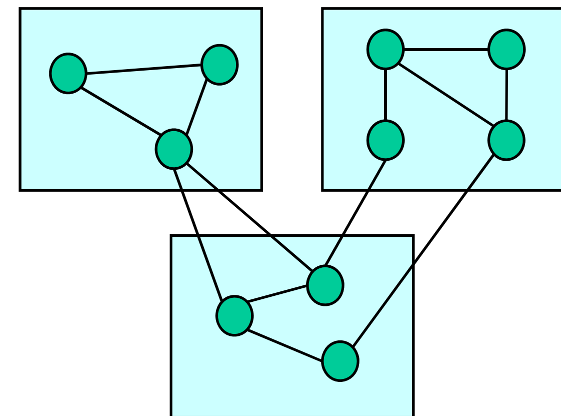
- Haphazard connections, probably grown over time
- No visible cohesive groups
- High coupling: high interaction between random parts
- Understand it all or nothing



10 parts, 13 connections

Modular System

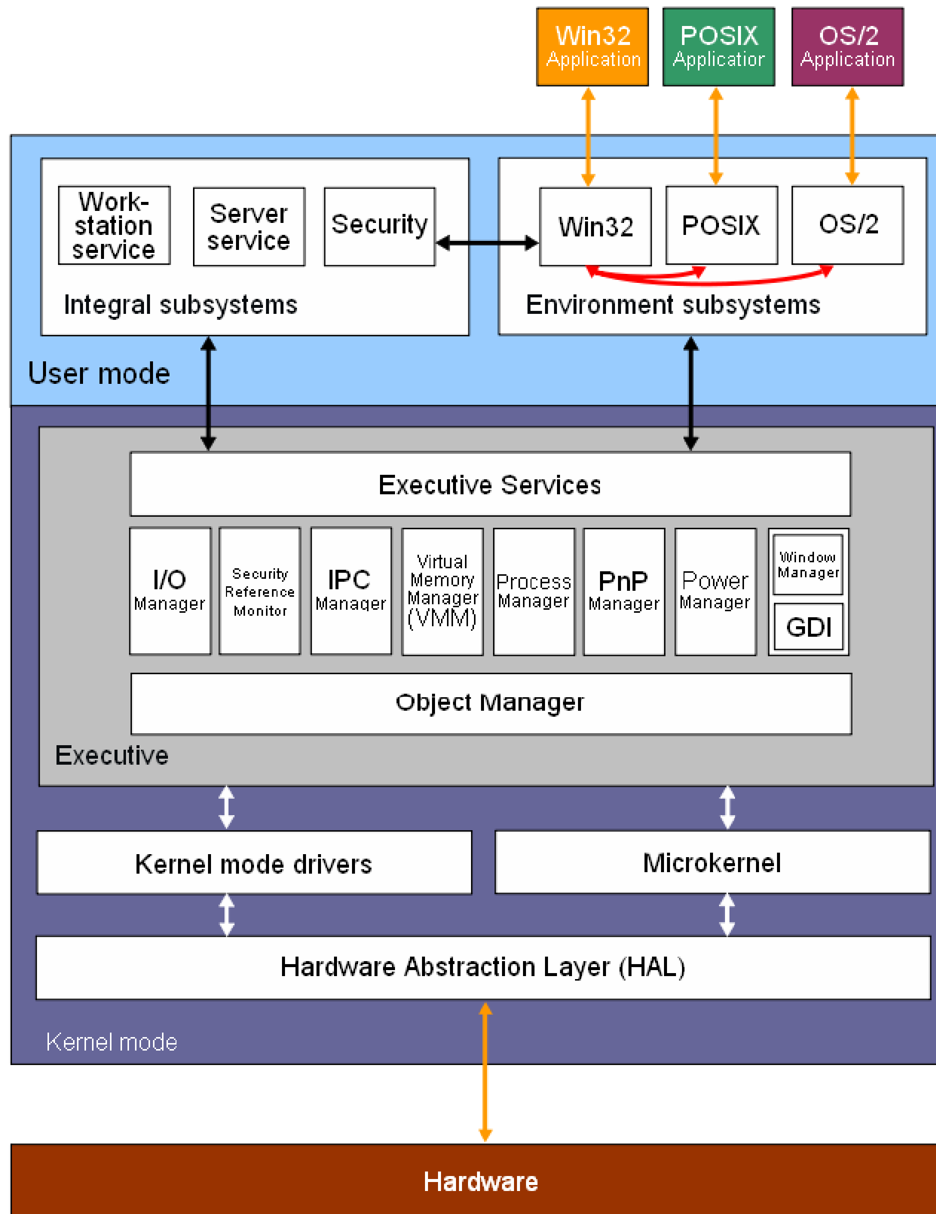
- High cohesion within modules
- Low coupling between modules
- Modules can be understood separately
- Interaction between modules is well-understood and thoroughly specified



10 parts, 13 connections, 3 modules

Layered Architecture

SE Software Engineering
The University of Auckland



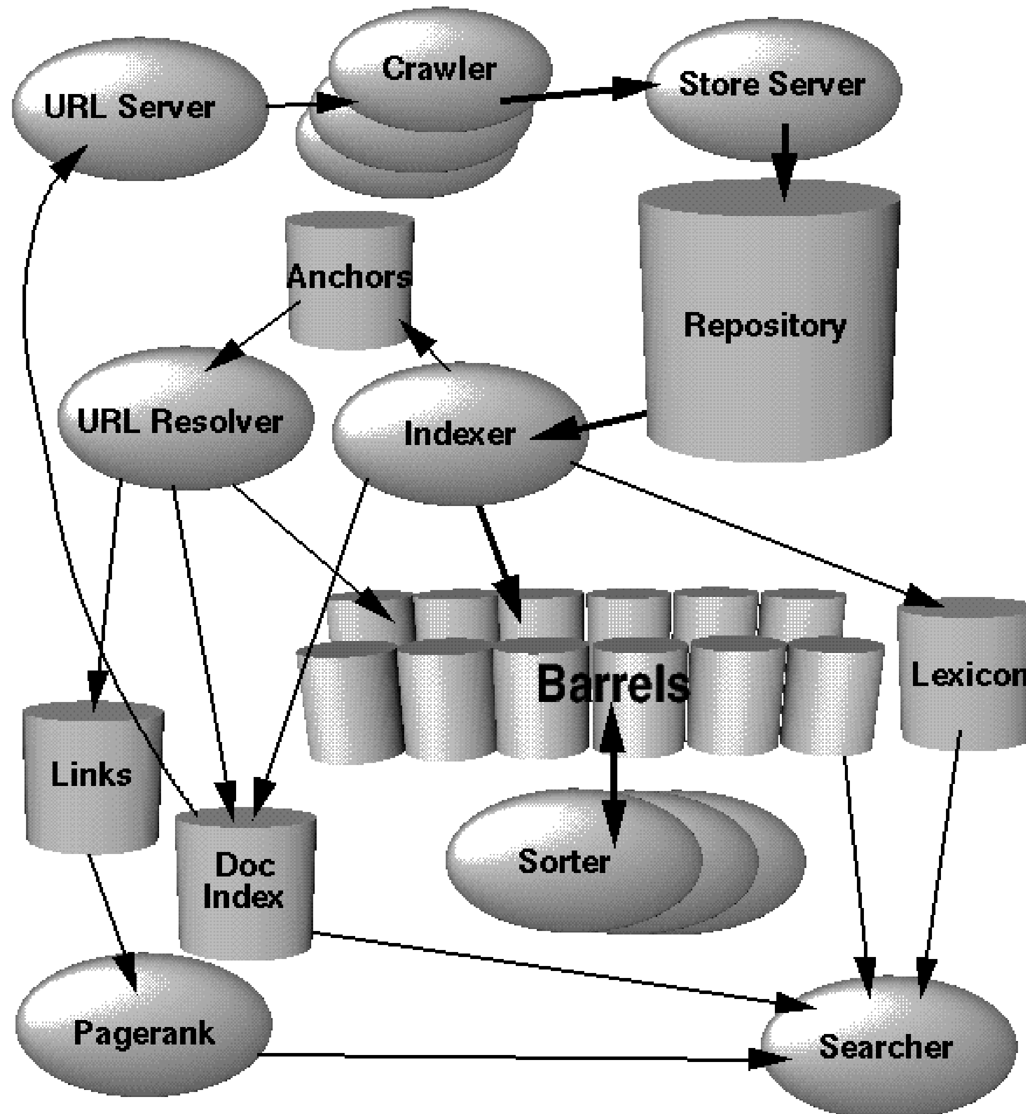
- Example: Windows OS
- Layered architecture:
  - Modules stacked onto each other
  - Often each level can only access the one below it
- Lowest level talks directly to hardware
- The higher, the more abstraction from concrete hardware

Separate Subsystems: Libraries and "Engines"

SE Software Engineering
The University of Auckland

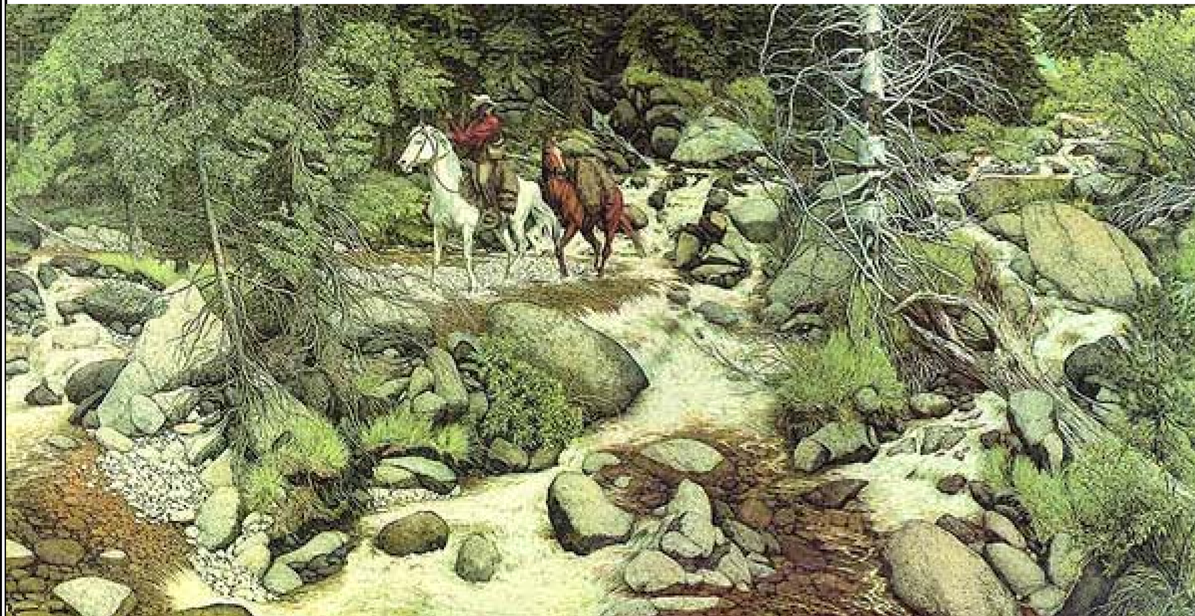
- Example: game
- Common problem: making them work together

Separate Subsystems: Parallel Processes



- Example: Google
- Massive data load requires massive **parallelism**
- Several modules working together concurrently and asynchronously
- Scalable architecture
- Modules can be optimized independently

Information Hiding



Out of sight,
out of mind.

Information Hiding

Software Engineering
The University of Auckland

- Hide information that does not need to be visible in order to use a class/module/program
- Too much information can be **confusing**: what is important for usage and what not?
- Too much information can lead to undesired **dependencies**
  - If internals are visible & accessible, someone might use/change them (e.g. create a "hack" to use something in an unintended manner)
  - If internals are changed then external code that relies on them might not work anymore
- Allowing only restricted access gives us more **flexibility**
  - Class/module/program can be (ex)changed without breaking other parts
  - Many design decisions can be hidden and the system design can evolve without collapsing

Scope

- Where we declare a variable determines where it can be accessed (i.e. its scope)
- Scope of instance variables > scope of local variables in methods > scope of local variables in statement blocks
- The scope of a variable should be **as small as possible**
- If a variable can be accessed where it should not be accessed: confusion and mistakes

```
class C {
  public int x;
  private int y;
  private int z;
  void m() {
    y = 0;
    for(z=0; z<10; z++)
      y += z;
    return y;
} }
```

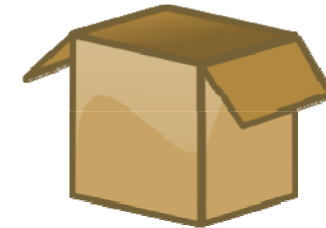
```
class C {
  public int x;


  void m() {
    int y = 0;
    for(int z=0; z<10; z++)
      y += z;
    return y;
} }
```

Access Modifiers

Can be used to control access to groups of parts

1. **`public:`**
   accessible everywhere



2. **`protected`** or no modifier (“default”):
   accessible in the same package

3. **`private`**:
   only accessible from within the class
   in which they are declared

General rule: expose parts only if necessary (same as for scope)

Limitations of access modifiers:

- Only for pre-defined groups
- Access rights only depend on **who** (what other class) wants access, not how they actually need to use it (e.g. only 1 method)

The Concept of Interfaces

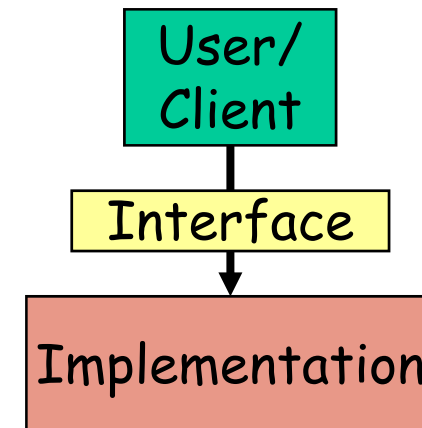
SE Software Engineering
The University of Auckland

There are different kinds of interfaces

- **User Interfaces**
  - Not just for software: any kind of tool
  - Usually it may change, sometimes it must not change
- **APIs**: important for programs that use them
- **Java `interfaces`**: important for classes that use other classes through them

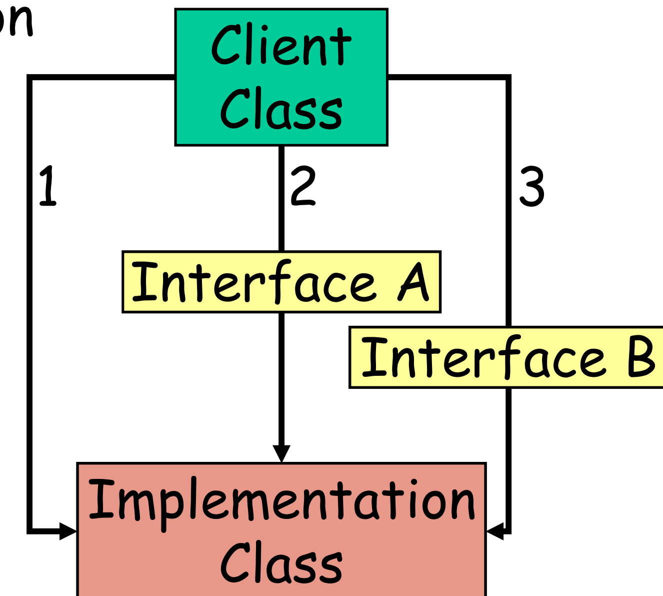
The intention is always the same:

- The interface defines and restricts **how** something can be used
- Users/clients perform operations only through the interface
- If the internal implementation changes, the users/clients do not have to change



Java Interfaces

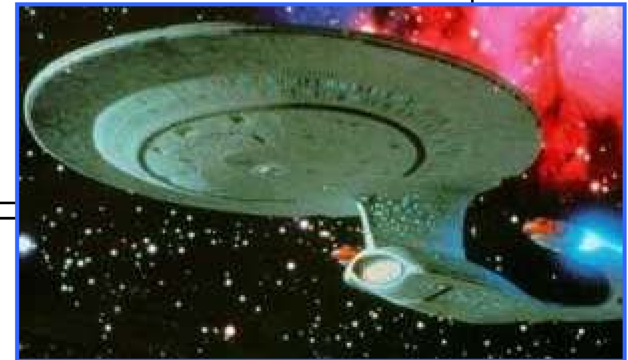
- Access modifiers can only control access depending on **who** uses a class/method/field
- With interfaces we can restrict access in a more flexible manner:
  - A class can implement several interfaces
  - Use a different interface depending on
    - **who** uses it (which other class)
    - **what** it is used for
- However: often using interfaces vs. accessing a class directly is a conscious decision
  - Programmers need to know that they should use interfaces
  - Programmers need to know which interfaces to use



1 or 2 or 3 ?

Interfaces Example

```
public class USSEnterprise
 implements Maintenance, SafeControl, FullControl {
   public void navigate(Point dest) { … }
   public void warpJump(Point dest) { … }
   public int checkSystems() { … }
   public void selfDestruct() { … }
}
```



```
interface Maintenance {
   public int checkSystems();
}
```

```
interface SafeControl extends Maintenance {
   public void navigate(Point dest);
   public void warpJump(Point dest);
}
```

```
interface FullControl extends SafeControl {
   public void selfDestruct();
}
```

Interfaces Example

SE Software Engineering
The University of Auckland

Scotty accessing the system:

```
Maintenance e = new USSEnterprise();
int status = e.checkSystems();
```

Spock accessing the system:

```
SafeControl e = new USSEnterprise();
e.warpJump(new Point(103, 789));
e.selfDestruct();
```

Borg accessing the system:

```
USSEnterprise e = new USSEnterprise();
e.selfDestruct();   // Boooom!
```

Access can be safely restricted by accessing through an appropriate interface.

- Choose an appropriate interface to access a class
- Accessing a class directly may lead to dependencies and other mistakes that could have been detected by the compiler

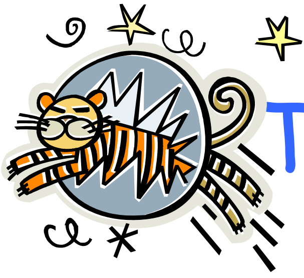
Enforcing Usage of Interfaces with Factories


```
protected class USSEnterprise
 implements Maintenance, SafeControl, FullControl {
   protected USSEnterprise() {}   ...    }
```

```
public class EnterpriseFactory {
   public static Maintenance getMaintenance() {
     return (Maintenance) new USSEnterprise();
   }
   public static FullControl getFullControl(String pw) {
     if(pw.equals("please"))
       return (FullControl) new USSEnterprise();
     else throw new RuntimeException("Alarm!!!");
   }
   // similar for SafeControl
}
```

Now access only possible through interfaces:

```
Maintenance e = EnterpriseFactory.getMaintenance();
int status = e.checkSystems();
```



Today's Summary

- **Modularity** means that a system is composed of self-contained parts (modules) with well-defined interfaces
  - Can be achieved by "**separation of concerns**"
  - Rule of thumb: "**low coupling, high cohesion**"
- **Information hiding** means that only the information is visible that is actually necessary to use something
  - Access is only possible through well-specified **interface**
  - Implementation **internals are hidden** and can be (ex)changed without breaking the system

Quiz

Software Engineering
The University of Auckland

1. What advantages do modular systems offer?
2. Why do we want modules to be lowly coupled?
3. What is the purpose of interfaces?