API Design

Workshop

Dr. O.S. van Roosmalen
Object-Oriented Software Engineering
Training and Consultancy
O.S.vanRoosmalen@chello.nl

API Design Workshop

- Some Theory:
  - API Definition
  - API Requirements
  - API Design Heuristics
- Some Practice
  - API Design Workshop
  - API Design Evaluation
  - API Design Workshop
  - API Design Evaluation
- Conclusions

API Definition

- Definition
  - An Application Programmer's Interface of a component or framework is the interface directly of relevance to a software engineer who develops applications based on the component or framework.
- The interface of a component or framework is more than the public operations and attributes of the realizing classes

API Aspects

- Operation and Attribute signatures
  - Classes and their public (or protected) operations and attributes
- Files (or in general persistent data)
  - File format
  - File Contents
- Environment variables and command-line options
- Text Messages
  - Anything a program reads or prints
- Protocols
- Behavior

API Motivation

- A component or framework with an API is offered to a third party to reduce the amount of work involved in system development by offering a reusable implementation of generic and often recurring functionality.

API Requirements

- Related Requirements:
  - Understandability
    - It must be easy for users to understand the purpose of the API as a whole as well as individual features (particularly often used ones)
    - However users of an API want to be “selectively clueless”
  - Ease of Use, discoverability
    - Common usage scenarios of the API must be easy to realize and must be well documented and easy to discover
  - Correctness
    - Users want to be able to rely on the correctness of the API implementation
  - Consistency
    - API must be based on a few concepts that are consistently applied
  - Preservation of Investment
    - Users must be convinced that it pays in the end to invest in learning to work with the API

API Pervasiveness

- API are abundant
  - Open source projects
  - IDEs
    - Eclipse
    - Netbeans
  - Libraries/frameworks
  - Platforms
    - Operating Systems
    - Middleware
    - Virtual machines

API's in Practice

- Preservation of investment implies that an API client has a long live time
- A good API is used by many (different) users.
  - It is impossible to understand and incorporate all the user's needs
- To make a good API the implementor must have experience with a wide group of users
- To make a bug-free and robust first (and second and third) time implementation is impossible

Conclusions:

- Obtaining a good API is an evolutionary process
- Public API's are forever – one (!?) chance to get it right

API Importance

- APIs can be among a company's greatest assets
  - Customers invest heavily: buying, writing, learning
  - Cost to stop using an API can be prohibitive
  - Successful public APIs capture customers
- Can also be among company's greatest liabilities
  - Bad APIs result in unending stream of support calls

Evolving APIs

- Requirements change over time: strategic evolution plan
  - One approach: make it from scratch
  - Other approach: enhance while going along
- The first version is never perfect: Incremental improvement
  - Bug fixes
  - Performance enhancements
  - Added functionality
- Whatever the changes: backward compatibility is essential

Backward compatibility

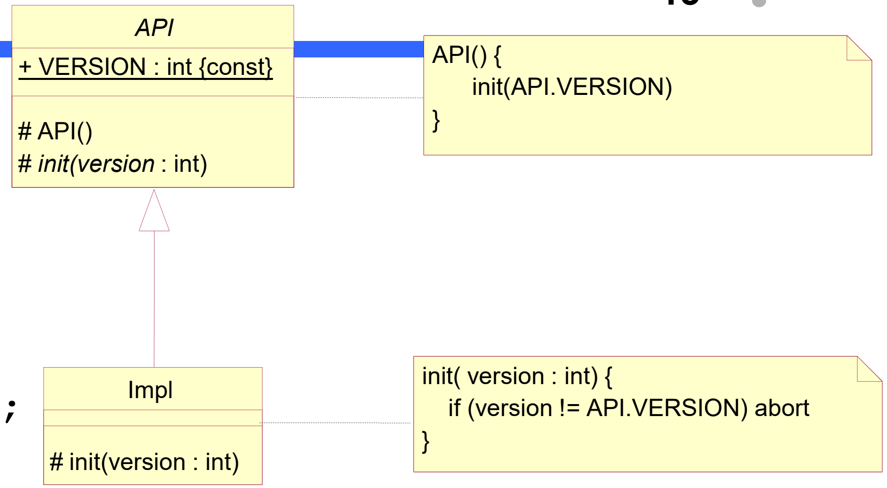
Three levels of compatibility

- Source compatibility
  - Client code written against old API compiles against new version
- Binary compatibility
  - Client code running against old API runs against new version
- Functional compatibility
  - Client code running correctly against old API runs correctly against new version

Incompatibility Examples

- Source compatibility:
  - Abstract class in API has more abstract operations in new version
  - Client will not implement these new operations and will not compile
- Binary compatibility
  - Can we run client code compiled with a newer/older version against a older/newer version?
  - Are the binaries obtained by compiling against two versions of the API the same? NO! In general byte code may differ.
  - See example next page!
- Functional compatibility
  - A bug in an older version may be considered a feature by a user
  - With newer version that removes the bug the client application has different behavior

Binary Incompatibility




```
public abstract class API {
   public static final int VERSION =1;
   protected API() {
        System.err.println("Initializing version " + VERSION);
        init(API.VERSION);
        System.err.println("Properly initialized: " +this);
   }
   protected abstract void init(int version)
                                     throws IllegalStateException;
}
```

Constant inserted at compile time

```
public class Impl extends API {
   protected void init(int version)throws IllegalStateException {
        if(version != API.VERSION)
           throw new IllegalStateException( "API version error!" );
   }
}
```

General Guide Lines

- Design an API Use-Case driven
  - Define typical abstract usage scenarios of the API
  - These scenario's provide easy clues for users on how to use the API
- Design API's consistently
  - Define a set of good practices for the API development team
- Make an API simple and clean
  - Common usage scenario's should be easy to implement
  - The offered interfaces refer to simple and clear abstractions
- Less is more
  - Only functionality required in usage scenario's should be exposed
- Design with evolution in mind
  - Anticipate on possible future requirements
  - Maintain backward compatibility

API Life Cycle (example)

- Private API
  - Not intended for use outside a particular part of a system
- Friend API
  - To be shared by different parts but not outside of a system
  - To be used by the same team and built at the same time
- Under Development
  - An incomplete contract, expected to become a stable API
  - Incompatible changes may appear in the future
- Stable
  - No incompatible future changes expected
- Official API
  - Stable API with published contract rolled out to third parties
- Deprecated
  - The API is likely to be replaced by one that provides better support

API Detailed Design Heuristics

- An operation is better than an attribute
- A factory class-operation is better than a constructor
- Make everything final
- Do not put setters where they do not belong
- Allow access only from friend code
- Avoid deep hierarchies

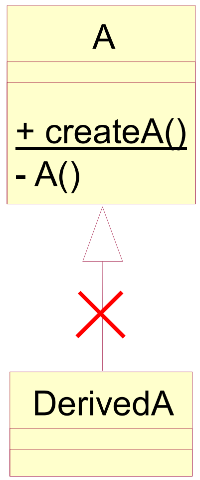
An operation is better than an attribute

- A getter can do much more than just getting the attribute
- An operation can be moved to a superclass without compromising binary compatibility
- Once an attribute is defined in a class it has to stay there

- Never expose attributes in an API, except for public static final primitive or string constants, enum values or immutable object references

A factory is better than a constructor

- Provide a public static operation for creating objects of the class
- Make constructor private

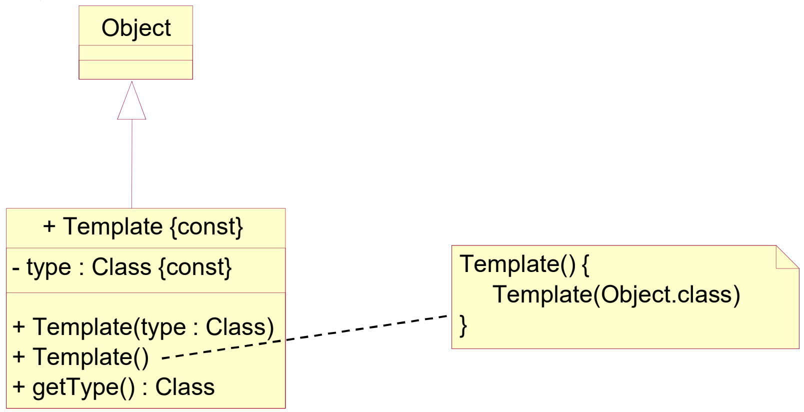


- Benefits
  - Users can not sub-class: solves a lot of evolution problems
  - One can create a cache or have pool of instances to return
  - The operation can be synchronized, preventing parallel access
  - One can return subclass instances of the return type

Example

- Home made pre-Java 1.5 type parameterized class

```
public final class Template extends Object {
   private final Class type;
   public Template(Class type) { this.type=type;}
   public Template() { this(Object.class); }
   public Class getType() { return type; }
}
```



- Want to migrate to Java 1.5

Java Details

- When a generic type is instantiated,
  - Java translates those types by a technique called *type erasure*
  - Compiler removes all information related to type parameters
  - Maintains binary compatibility between applications that use generics with applications that were created before generics.
- Erasure
  - A<B> is translated to a single class A, which is called the *raw type*
  - No run-time information on type of Object a generic class is using
  - The following operations are not possible:

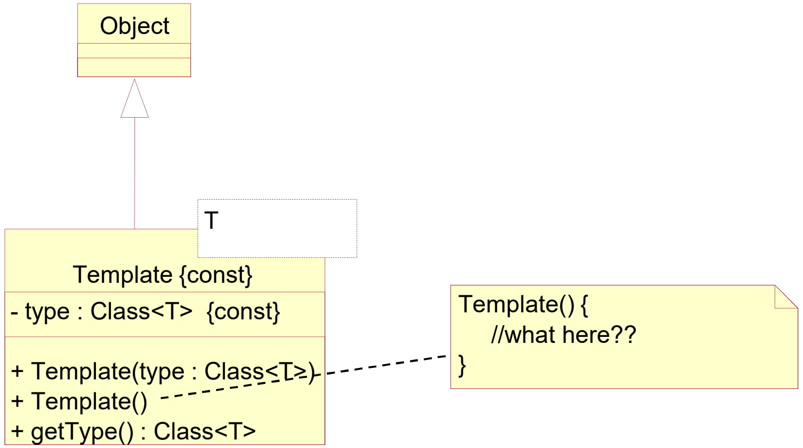
```
public class MyClass<E> {
   public static void myMethod(Object item) {
       if (item instanceof E) { //Compiler error
           ...
       }
       E item2 = new E(); //Compiler error
       E[] iArray = new E[10]; //Compiler error
       E obj = (E)new Object(); //Unchecked cast warning
   }
}
```

Example

- Java 1.5 type parameterized class

```
public final class Template<T> extends Object {
   private final Class<T> type;
   public Template(Class<T> type) { this.type=type;}
   public Template() { this(Object.class); }  // what here??
   public Class<T> getType() { return type; }
}
```

- We like to get instance of Template<Object>
- Deprecate constructor?

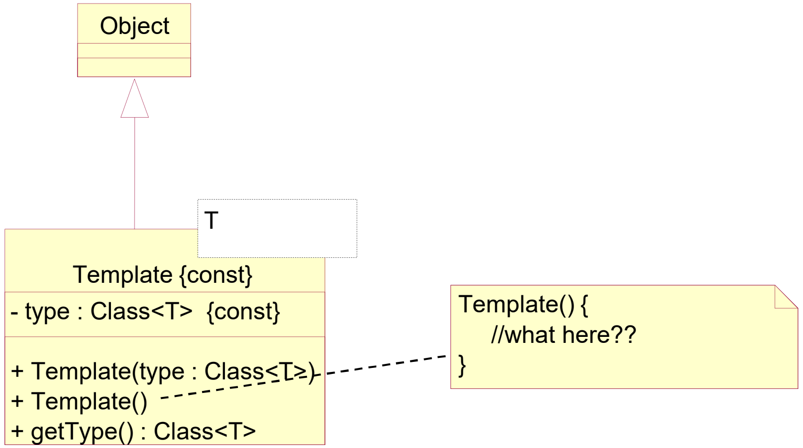


Example

- Java 1.5 type parameterized class

```
public final class Template<T> extends Object {
   private final Class<T> type;
   public Template(Class<T> type) { this.type=type;}
   public Template() { this(Object.class); }  // what here??
   public Class<T> getType() { return type; }
}
```

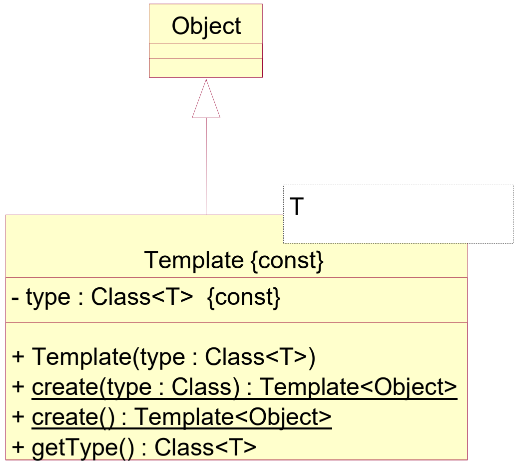
- We like to get instance of Template<Object>
- Deprecate constructor?



Problem Avoided

- Java pre-1.5 class

```
public final class Template extends Object {
   private final Class type;
   private Template(Class type) { this.type=type;}
   public static Template create(Class type){ return new Template(type); }
   public static Template create() { return new Template(Object.class);}
   public Class getType() { return type; }
}
```

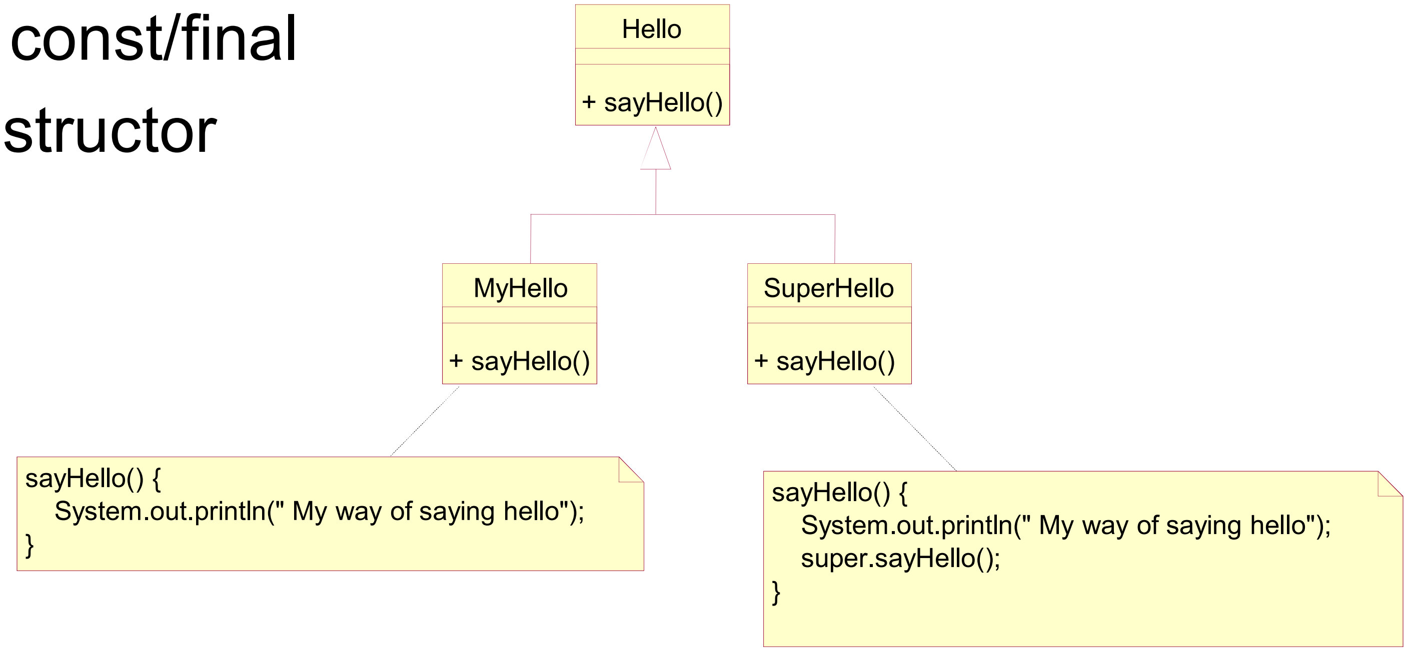


- Is replaced by generic class

```
public final class Template<T> extends Object {
   private final Class<T> type;
   public Template(Class<T> type) { this.type=type;}
   public static Template<Object> create(Class type){return new Template(type);}
   public static Template<Object> create(){ return new Template(Object.class);}
   public Class<T> getType() { return type; }
}
```

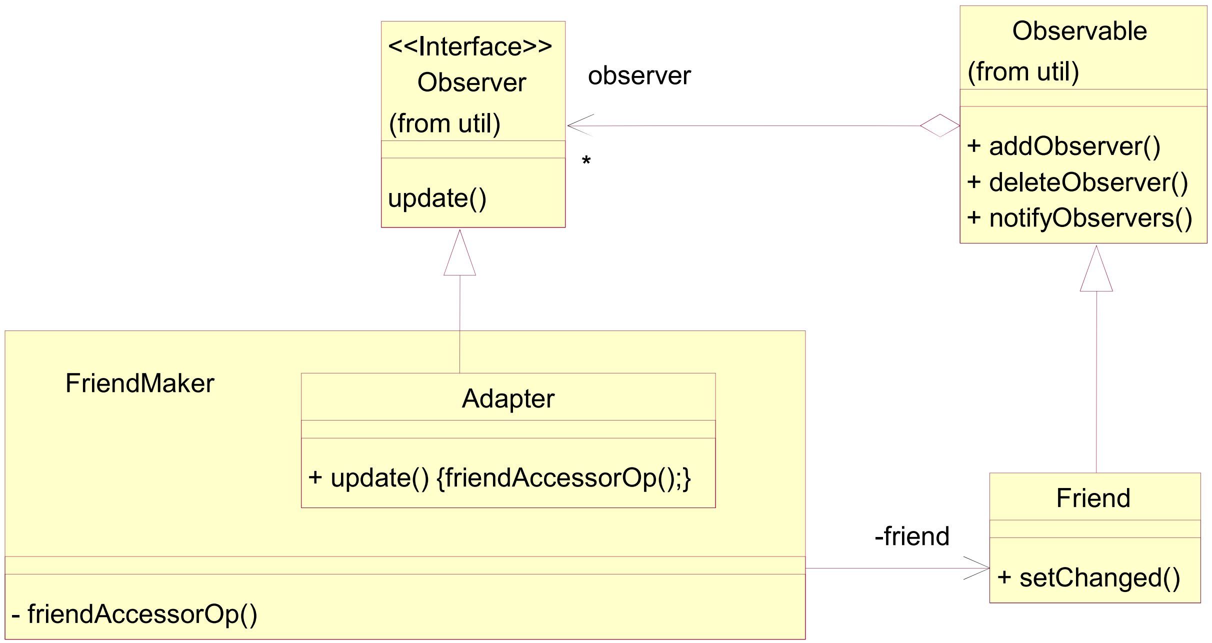
Make everything final

- A class offered by the API can be overridden.
- You have maintain support for three usages of operations
  - Direct use
  - Override
  - Override and call back old operation
- Make class const/final
- Or hide constructor



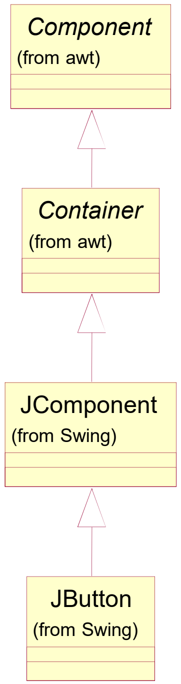
Allow access only from friend code

- Friend declaration in C++
- Listener/Observer using inner classes that access private functionality
  - A way of providing selective access to private operations
  - addObserver() grants access to friends via private operation



Avoid deep hierarchies

- Use Inheritance only for polymorphism
- Use simple concepts: deep hierarchies are not conceptually simple
- Export simple interfaces and abstract classes
- If you export a specialization ask the question:
  - Is it a specialization for the sake of the API user or for the convenience of the API implementor



API Global Design Heuristics

- Use Modular Architecture
- Separate API’s for Clients and Providers
- Beware of using other API’s

API Design Workshop Intro

- The API Design Workshop is a form of contest
- It is aimed at teaching participants about the evolution problems related to writing an API.
- It consists of the following steps:
  - A simple task to write an API for a given problem
  - Results will be evaluated
    - The solutions will be presented
    - Participants will give their judgments on the API
  - An extension will be defined to be performed on the API
  - Results will again be evaluated.
    - The solutions will be presented.
    - Participants will suggest a test that works for previous version of an API, but does not work in the new one.

Deliverables

- A description of the API (UML and Java).
- A design of the implementation (UML)
- A java implementation that runs against the test cases.
- The test case implementation

  Three test cases:

  - Create a circuit to evaluate "x1 and x2" and then verify that its result is false for input (false, true) and it is true for input (true, true).
  - Create a circuit to evaluate "(x1 and x2) or x3" and then verify that its result is false for input (false, true, false) and it is true for input (false, false, true).
  - Create a circuit to evaluate "(x1 or not(x1))" and then verify that its result is true for all values of x1

API Design Workshop Schedule Day 1

- 09:00 – 10:00 Lecture
- 10:00 API Assignment version 1
- 10:00 – 12:00 API design
- 12:30 – 14:30 API implementation and testing
- 14:30 Exchange of results version 1
- 14:30 – 15:30 Presentations
- 15:30 API Assignment version 2
- 15:30 – 17:30 API design version 2

API Design Workshop Schedule Day 2

- 09:00 – 11:00 API implementation and testing
- 11:00 Exchange of results version 2
- 11:00 – 17:30 Breaking the other's API's.

API Design Workshop Schedule Day 3

- 09:00 – 10:30 Breaking the other's API's.
- 10:30 – 11:30 Scoring
- 11:30 – 12:30 Lecture