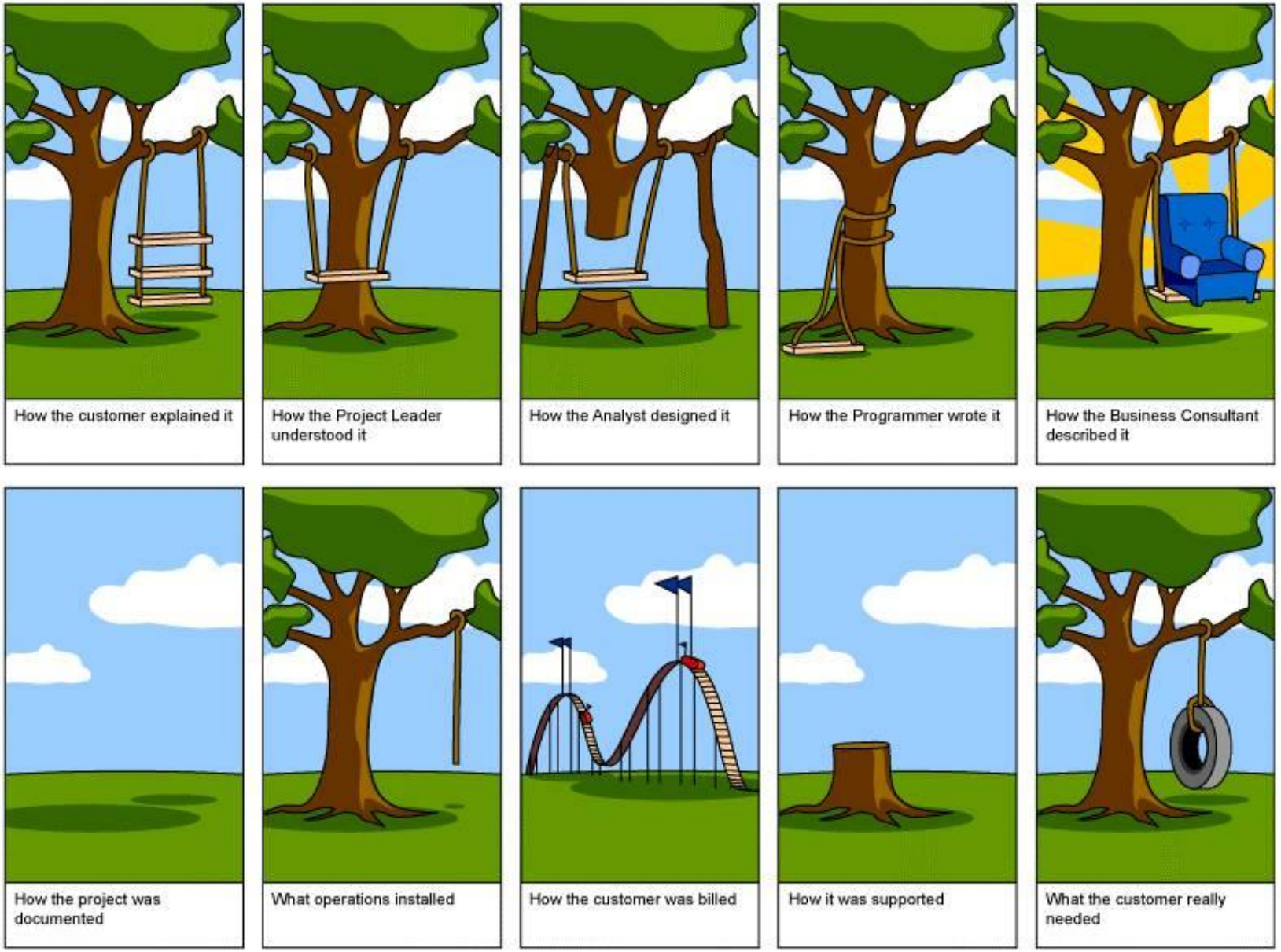
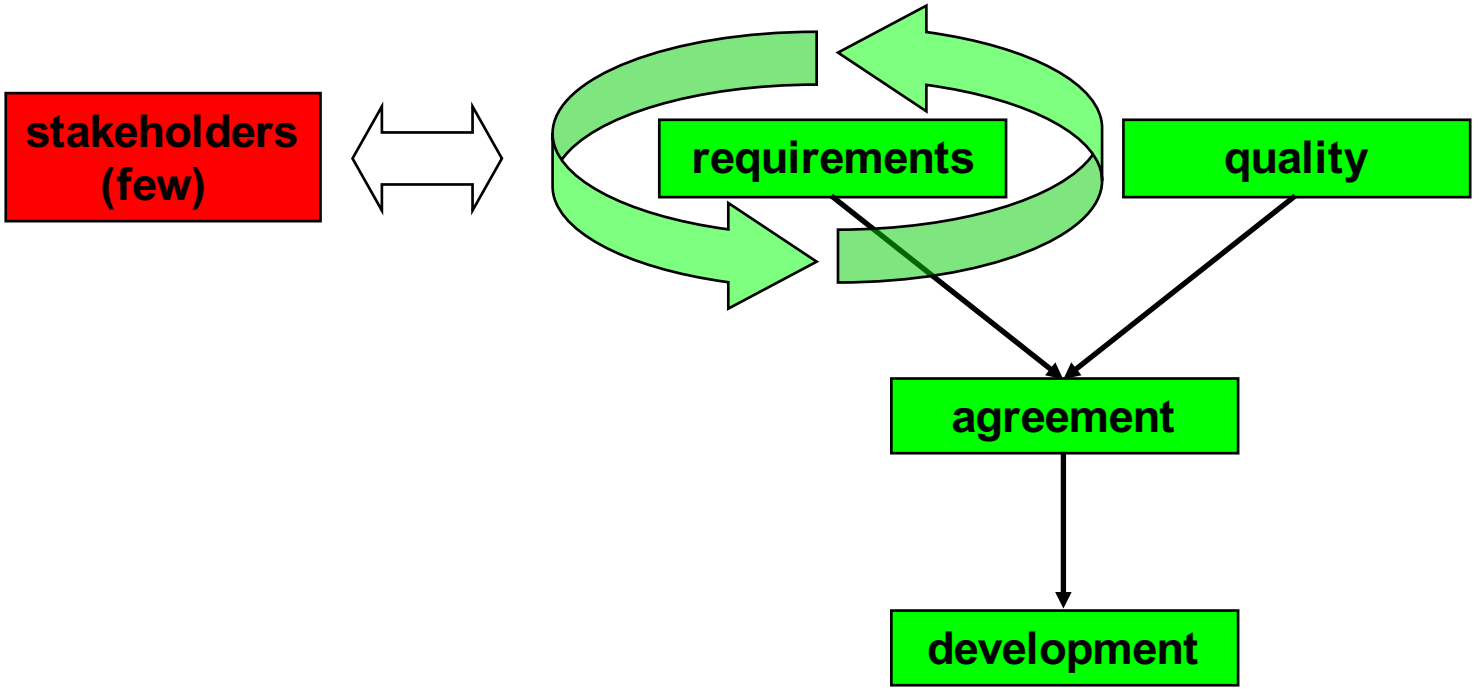
Politecnico
di Milano

Software Design &
Software Architecture

Software design



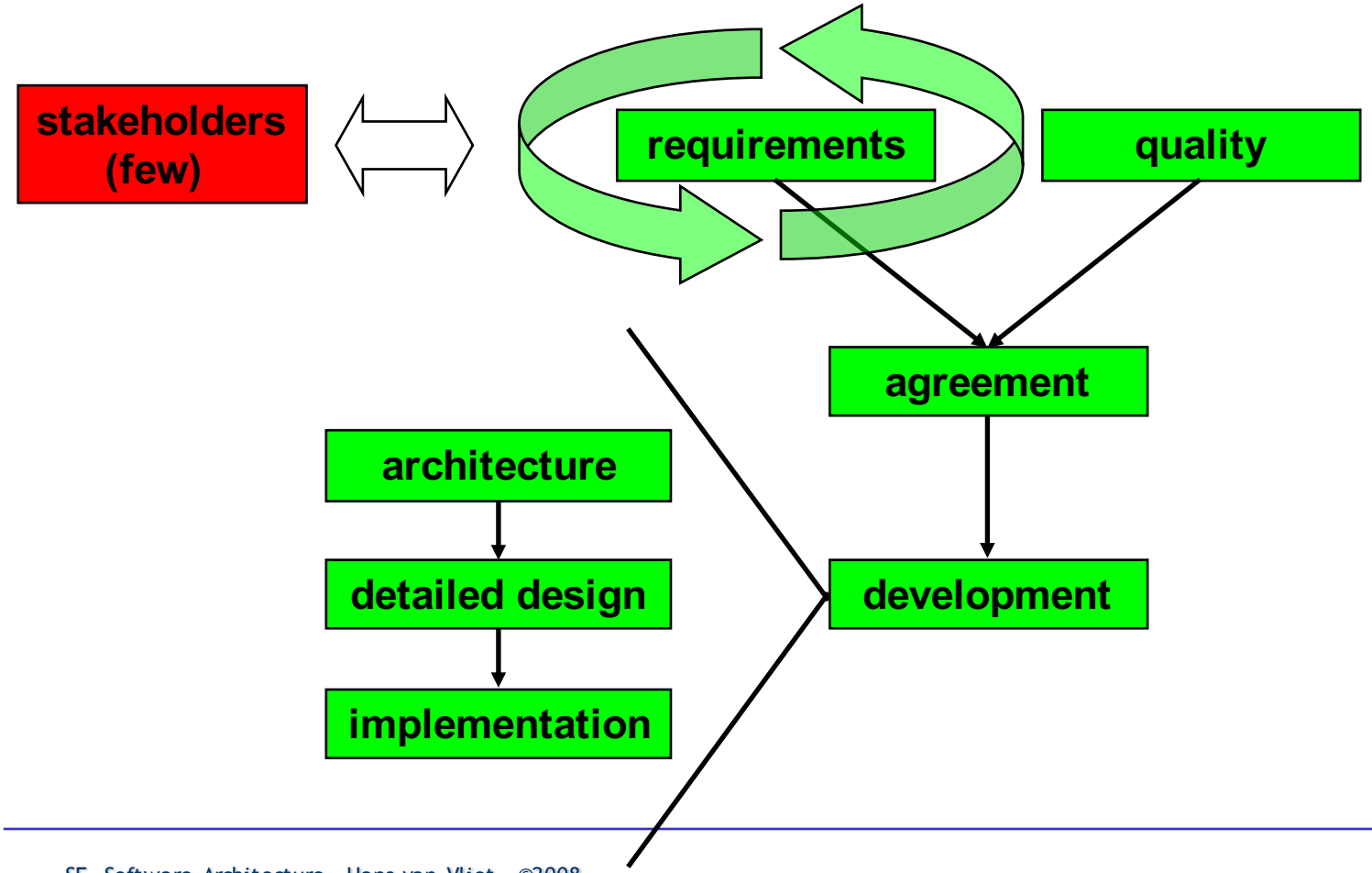
Pre-architecture life cycle



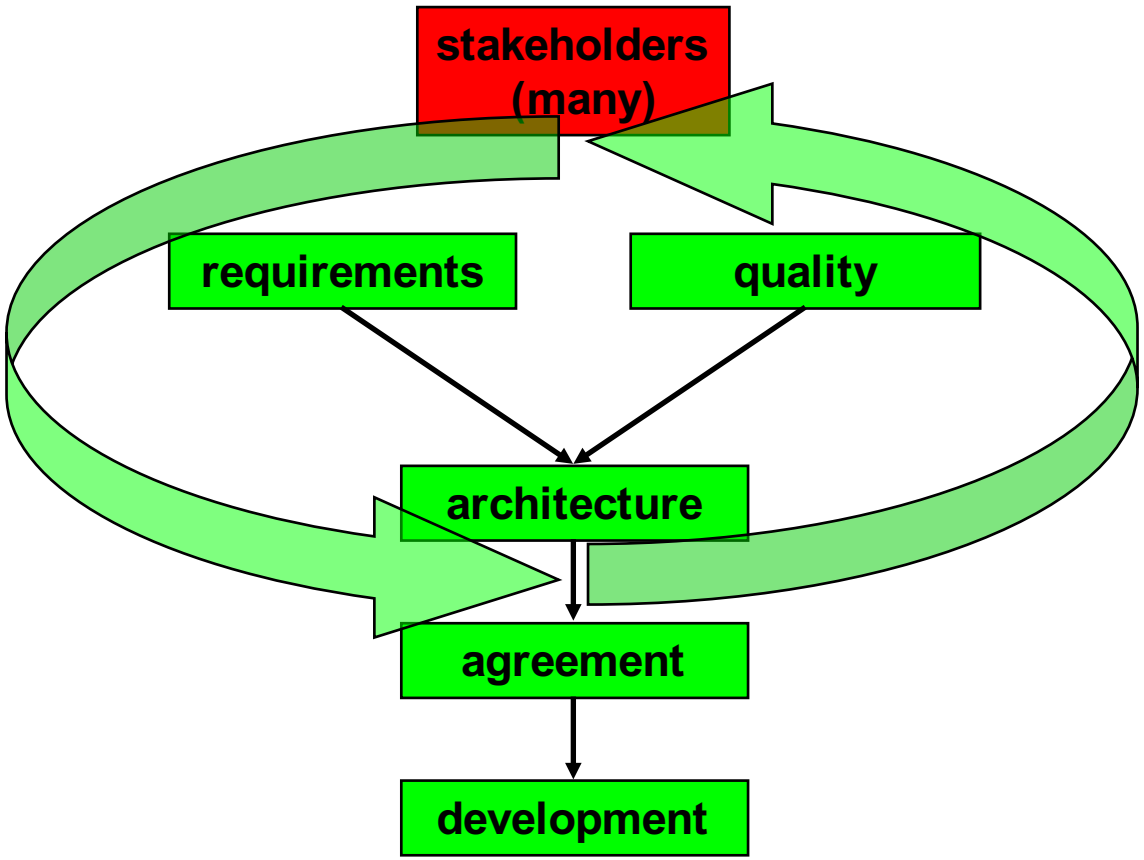
Characteristics

- Iteration mainly on functional requirements
- Few stakeholders involved
- No balancing of functional and quality requirements

Adding architecture, the easy way



Giving a more crucial role to the architecture



Characteristics

- Iteration on both functional and quality requirements
- Many stakeholders involved
- Balancing of functional and quality requirements

Why Is Architecture Important?

- Architecture is the vehicle for stakeholder communication
- Architecture manifests the earliest set of design decisions
  - Introduces constraints on implementation
  - Dictates organizational structure
  - Inhibits or enables quality attributes
- Architecture is a transferable abstraction of a system
  - Product lines share a common architecture
  - Allows for template-based development
  - Basis for training

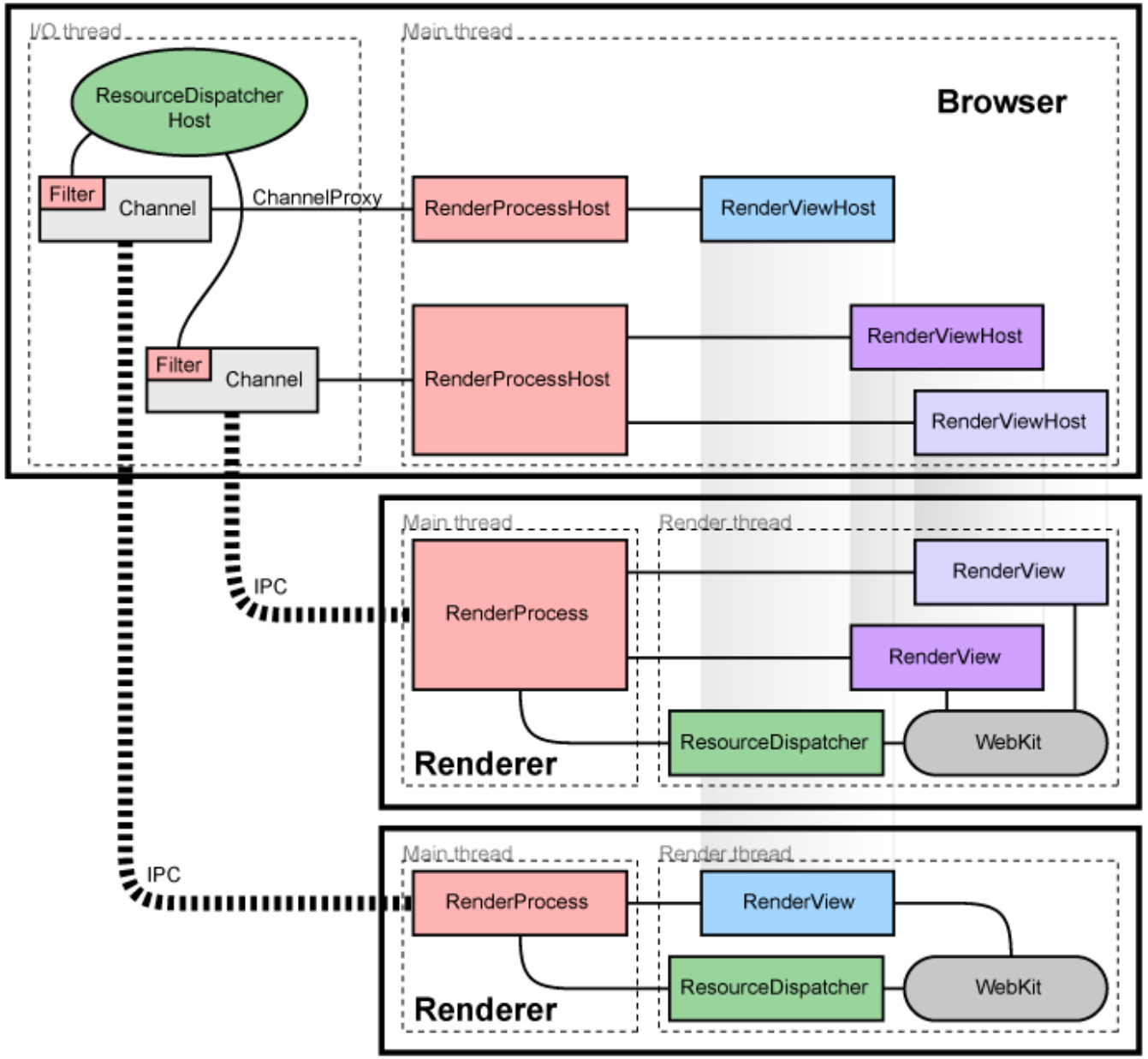
Software architecture, definition (1)

The architecture of a software system defines that system in terms of computational components and interactions among those components.

(from Shaw and Garlan, *Software Architecture, Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.)

Example – Chromium Architecture

https://www.chromium.org/developers/design-documents/multi-process-architecture



Software Architecture, definition (2)

The software architecture of a system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

(from Bass, Clements, and Kazman, *Software Architecture in Practice*, SEI Series in Software Engineering. Addison-Wesley, 2003.)

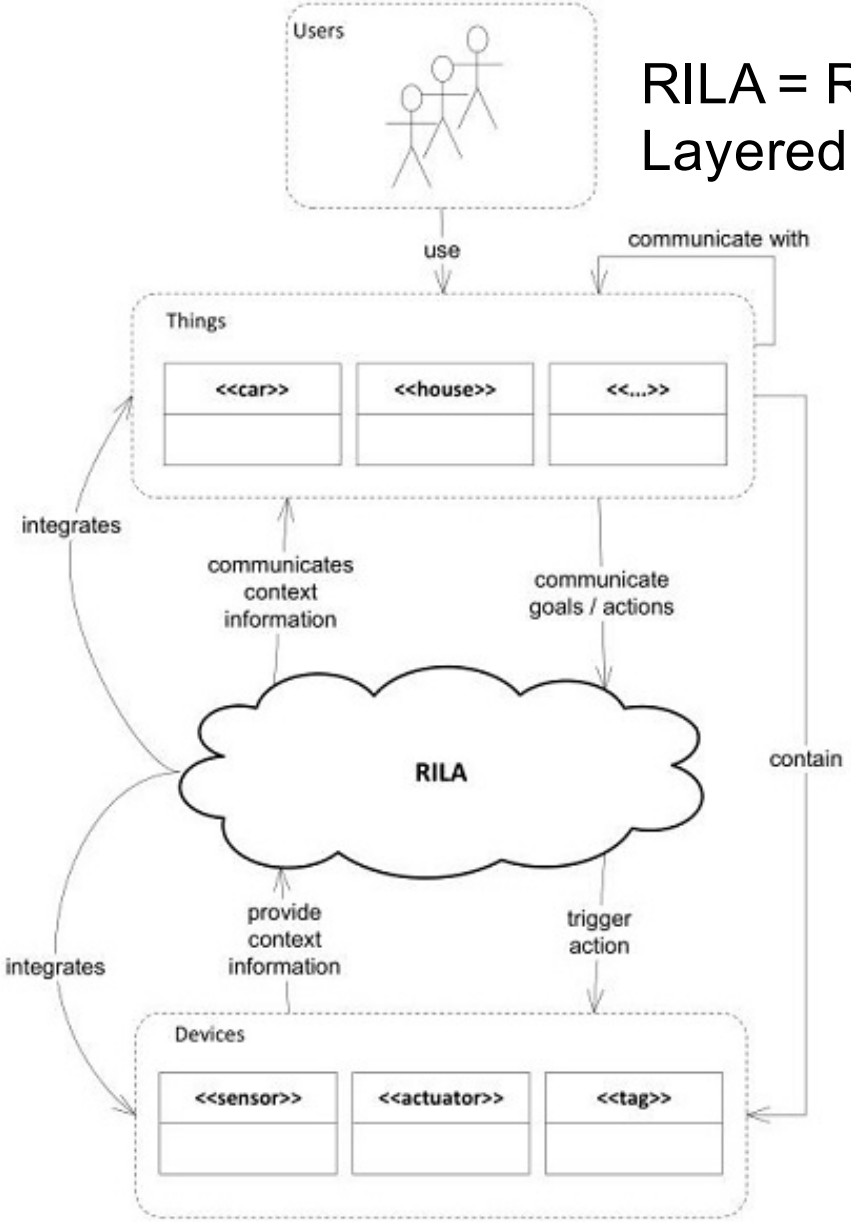
Software Architecture

- Important issues raised in this definition:
  - multiple system structures;
  - externally visible (observable) properties of components.

- The definition does not include:
  - the process;
  - rules and guidelines;
  - architectural styles.

Example – an Internet of Things reference architecture (1)

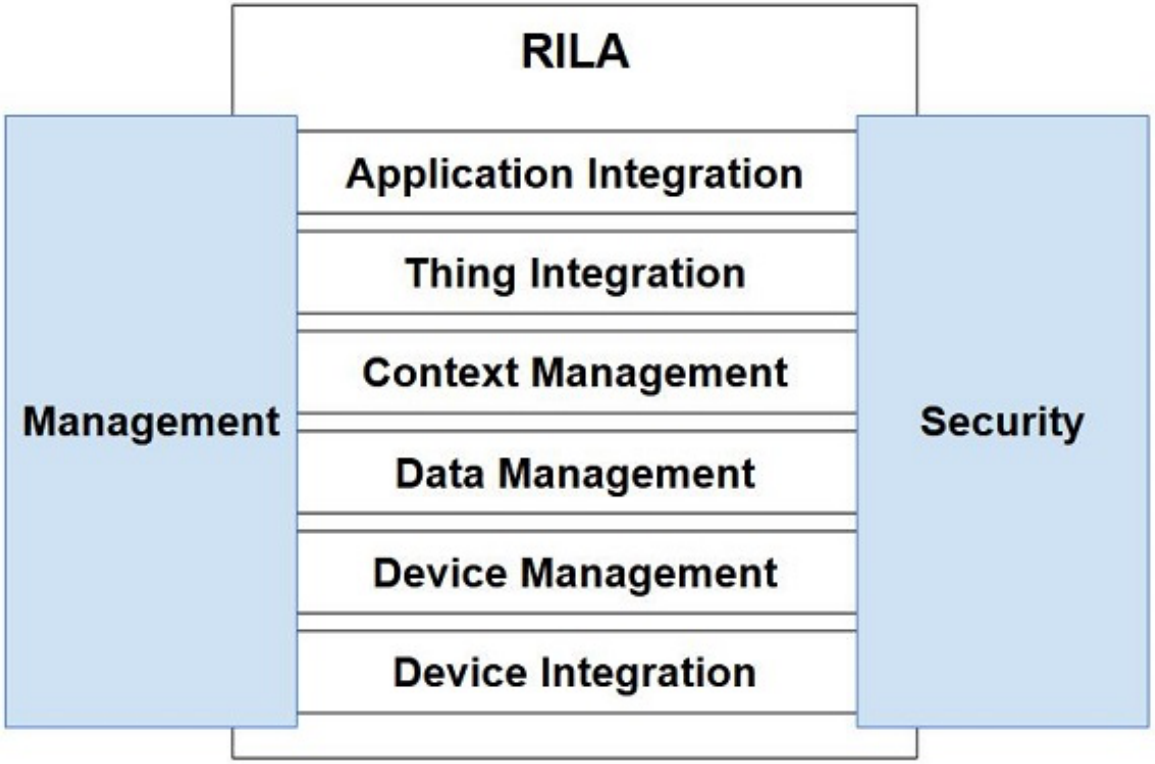
https://www.infoq.com/articles/internet-of-things-reference-architecture



RILA = Reference IoT Layered Architecture

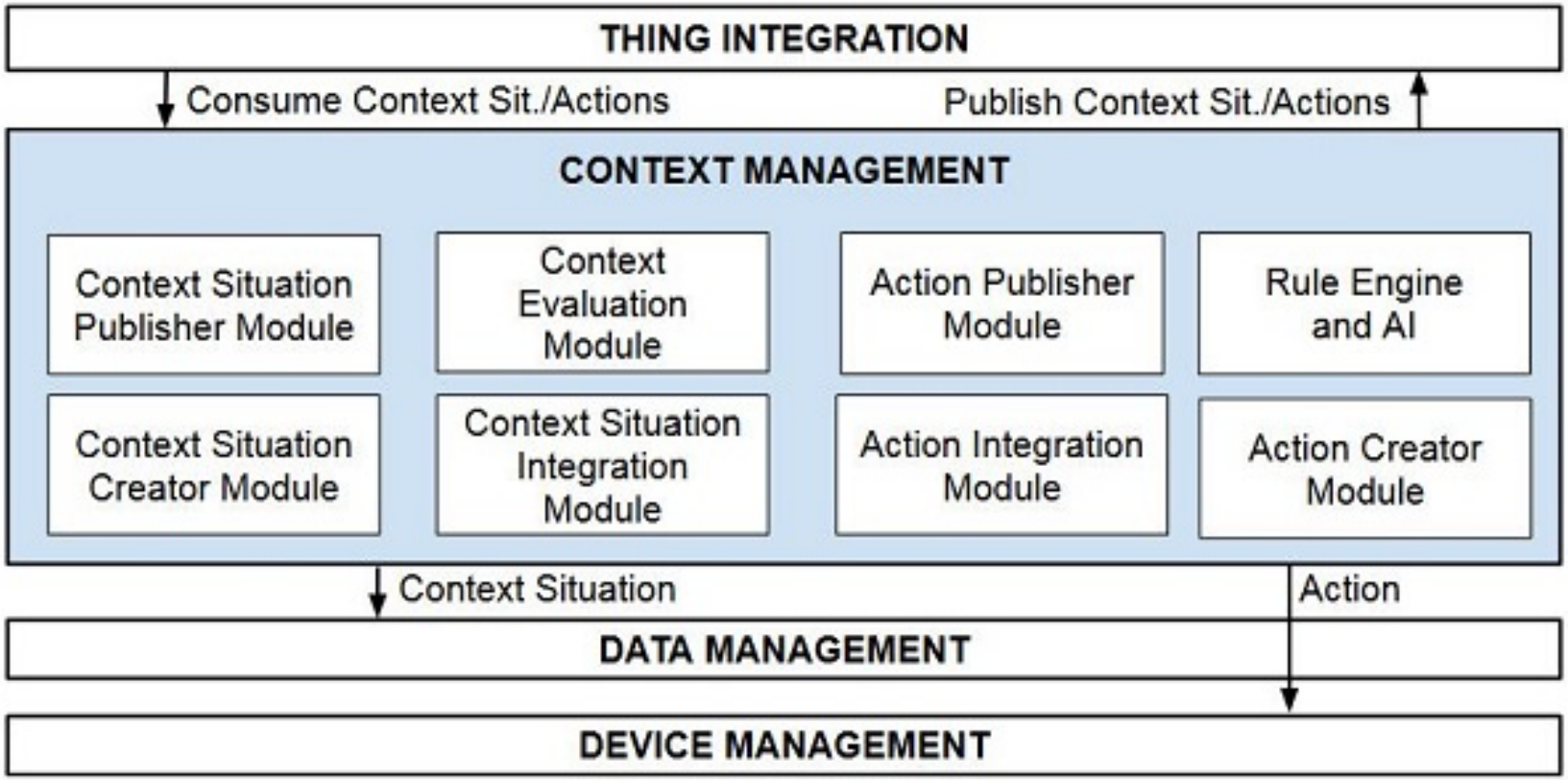
Example – an Internet of Things reference architecture (2)

https://www.infoq.com/articles/internet-of-things-reference-architecture



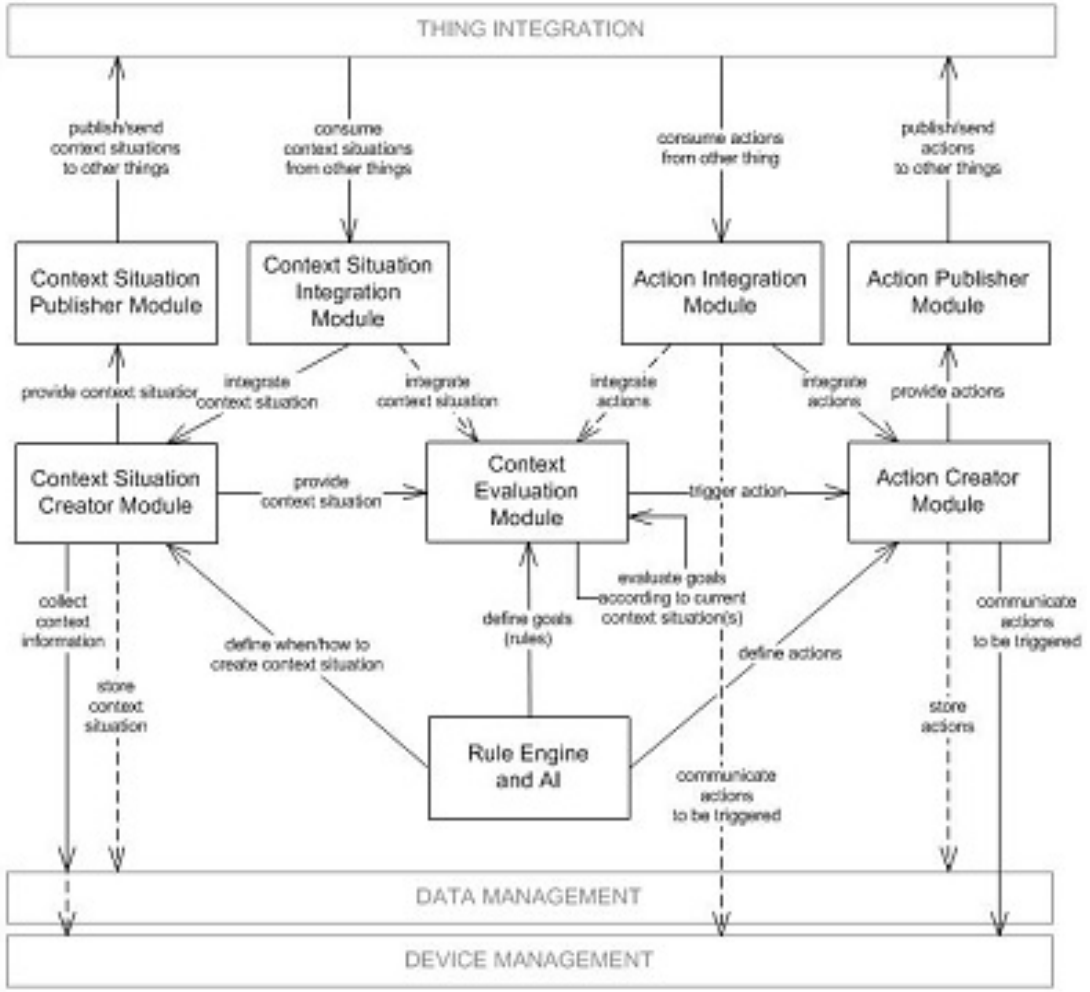
Example – an Internet of Things reference architecture (3)

https://www.infoq.com/articles/internet-of-things-reference-architecture



Example – an Internet of Things reference architecture (4)

https://www.infoq.com/articles/internet-of-things-reference-architecture



Politecnico
di Milano

The architecture is intrinsic to our software!

A simple example: a trivial prototype of a "banking application" - Account


```
import java.rmi.*;
public interface Account extends Remote {
        public float balance() throws RemoteException;
}

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class AccountImpl extends UnicastRemoteObject implements
Account {
        private float _balance;
        public AccountImpl(float balance) throws RemoteException {
                _balance = balance;
        }
        public float balance() throws RemoteException {
                return _balance;
        }
}
```

A simple example: a trivial prototype of a "banking application" - AccountFactory


```
import java.rmi.*;
public interface AccountFactory extends Remote {
        public Account createAccount(String userName, int balance)
                throws RemoteException;
        public int getLoad() throws RemoteException;
}
import java.util.*; import java.net.InetAddress; import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class AccountFactoryImpl extends UnicastRemoteObject
implements AccountFactory {
        private String hostName;
        private int load;
        public AccountFactoryImpl(String _hostName)
        throws RemoteException {
                hostName = _hostName;    load = 0;
        }
```

A simple example: a trivial prototype of a "banking application" - AccountFactory


```
public Account createAccount(String userName, int balance)
throws RemoteException {
        Account account = new AccountImpl(balance);
        try {
         Naming.rebind("//"+hostName+"/Account"+userName, account);
         System.out.println("Created " + userName +
             "'s account: " + account);
        } catch (Exception e) {        e.printStackTrace();        }
         load++;
        return account;
}
public int getLoad() throws RemoteException {    return load;  }
```

A simple example: a trivial prototype of a "banking application" - AccountFactory


```
public static void main(String[] args) {
        String _hostName;
        try {
         System.setSecurityManager(new RMISecurityManager());
        _hostName = InetAddress.getLocalHost().getHostName();
        AccountFactoryImpl server = new
                AccountFactoryImpl(_hostName);
        Naming.rebind("//"+_hostName+"/AccountFactory", server);
        System.out.println("Factory bound on "+_hostName);
        } catch (Exception e) {      e.printStackTrace();    }
}
}
```

A simple example: a trivial prototype of a "banking application" - AccountManager


```
import java.rmi.*;
public interface AccountManager extends Remote {
        public Account open(String name) throws RemoteException;
}
import java.util.*; import java.net.InetAddress; import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class AccountManagerImpl extends UnicastRemoteObject
implements AccountManager {
        private Dictionary _accounts = new Hashtable();
        private String hostName;
        private Vector factoryList;
        public AccountManagerImpl(String _hostName)
                throws RemoteException {
                hostName = _hostName;
                factoryList = new Vector();
        }
```

A simple example: a trivial prototype of a "banking application" – AccountManager


```
public void addFactory(String hostName){
        try {       AccountFactory fct = (AccountFactory)
                    Naming.lookup("//"+hostName+"/AccountFactory");
                    factoryList.addElement(fct);
        } catch (Exception e) {
                    System.out.println("Error: Factory unreachable");
                    e.printStackTrace();
        }
}
```

A simple example: a trivial prototype of a "banking application" - AccountManager


```
public AccountFactory selectFactory(){
        int minLoad, load;    int index=0;    AccountFactory f;
        try {       Enumeration e = factoryList.elements();
                    f = (AccountFactory) e.nextElement();
                    minLoad = f.getLoad();
                    index=factoryList.indexOf(f);
                    while(e.hasMoreElements()){
                                f = (AccountFactory) e.nextElement();
                                load = f.getLoad();
                                if(minLoad > load){
                                            minLoad = load;
                                            index=factoryList.indexOf(f);
        }     }    } catch (Exception e){
                                e.printStackTrace();    }
        return (AccountFactory) factoryList.elementAt(index);
}
```

A simple example: a trivial prototype of a "banking application" - AccountManager


```
public Account open(String name) throws RemoteException {
        AccountFactory factory;
        /* ...Security checks... */
        Account account = (Account) _accounts.get(name);
        if(account == null) {      factory = selectFactory();
        try {
                account = factory.createAccount(name, 0);
                _accounts.put(name, account);
        } catch (Exception e) {       e.printStackTrace();     }    }
        return account;
}
```

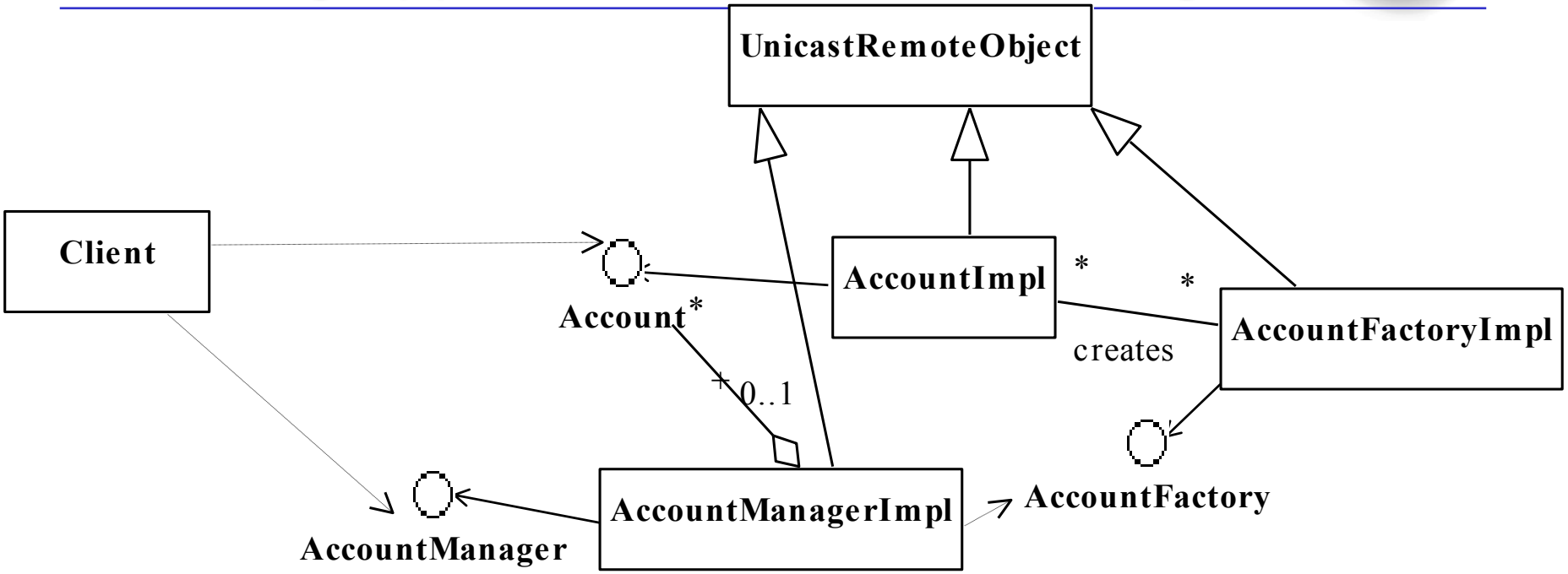
A simple example: a trivial prototype of a "banking application" - AccountManager


```
public static void main(String[] args) {
        String _hostName;
        try {
          System.setSecurityManager(new RMISecurityManager());
          _hostName = InetAddress.getLocalHost().getHostName();
          AccountManagerImpl server = new AccountManagerImpl(_hostName);
          server.addFactory(args[0]);
          Naming.rebind("//"+_hostName+"/AccountManager", server);
          System.out.println("Server bound on "+_hostName);
        } catch (Exception e) {      e.printStackTrace();    }
}}
```

A simple example: a trivial prototype of a "banking application" - Client

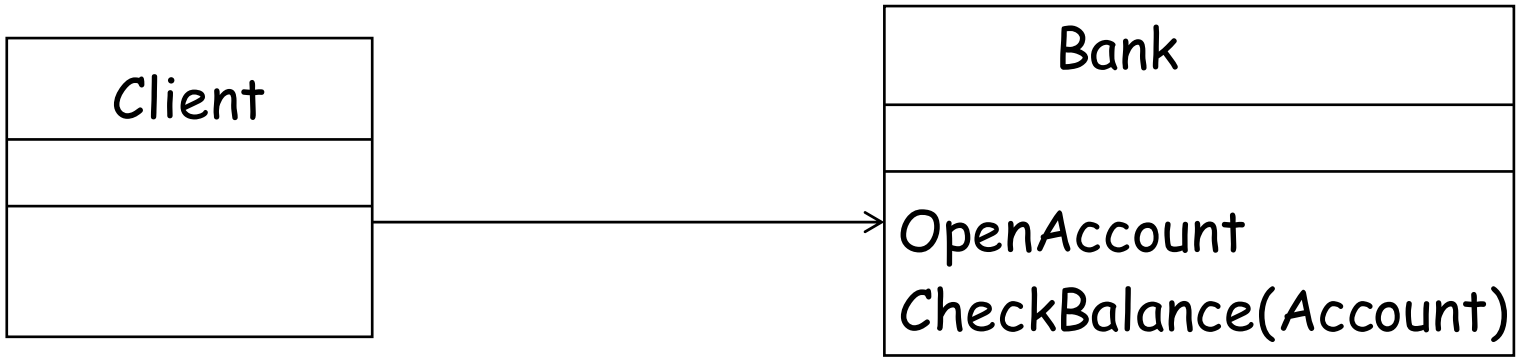

```
import java.rmi.*;
public class Client {
        public static void main(String[] args) {
        try {
                System.setSecurityManager(new RMISecurityManager());
                System.out.println("Looking up server...");
                AccountManager server = (AccountManager)
                Naming.lookup("//"+args[0]+"/AccountManager");
                System.out.println("Server bound...");
                String name = "Jack B. Quick";
                Account account = server.open(name);
                float balance = account.balance();
                System.out.println ("The balance in " + name +
                        "'s account is $" + balance);
        } catch (Exception e) {     e.printStackTrace();   }  } }
```

A simple example: a trivial prototype of a "banking application" - Low level design view



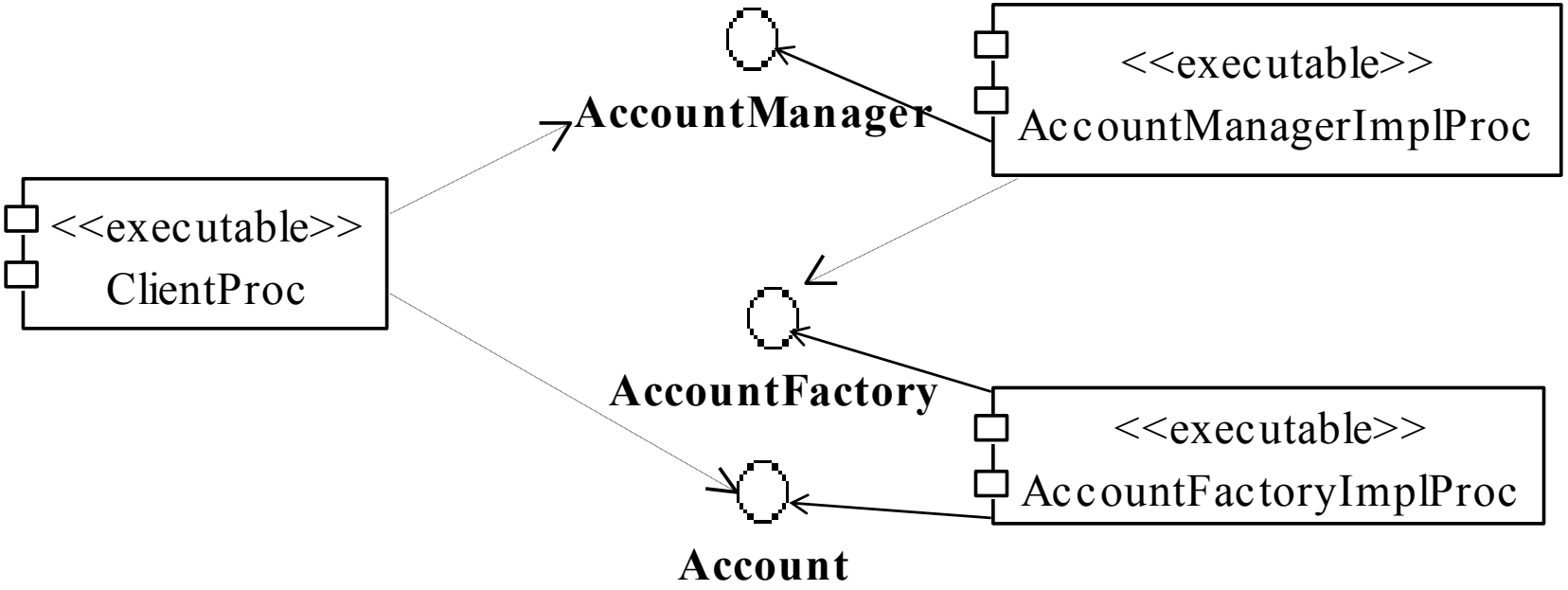
- What info do I get from this?
  - It uses remote objects
  - Client uses two interfaces
  - The other interface is used by AccountManager

A simple example: a trivial prototype of a “banking application” - Higher level view

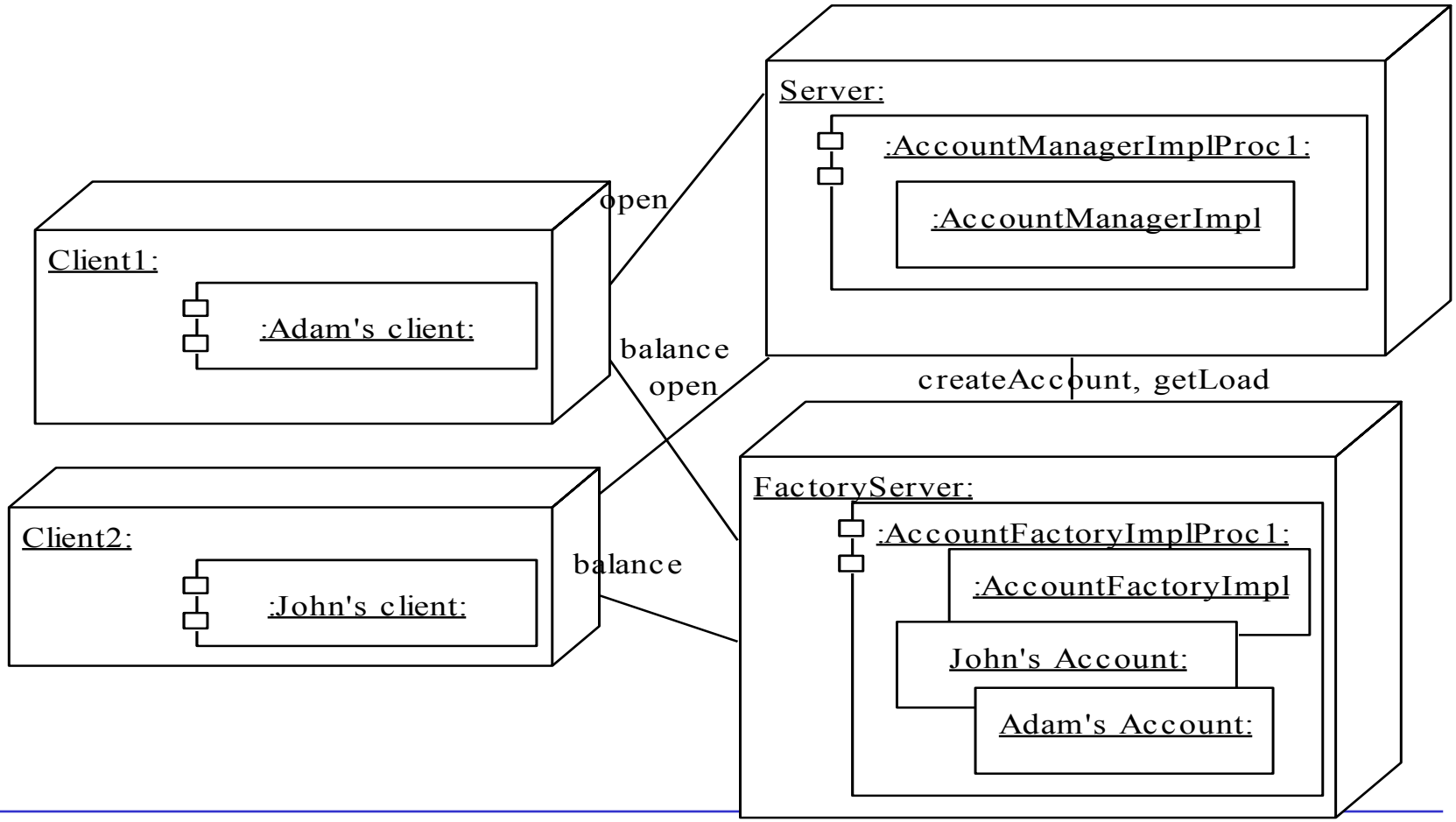


- A Bank Server is able to receive from clients two requests
  - Open account: it returns an account identifier. If the account does not exist, the system creates it
  - Balance: it checks the balance of a previously opened account
- Other details are made transparent:
  - The factory
  - The distinction between services offered by the manager and those offered by the account

A simple example: a trivial prototype of a “banking application” - Executables (*deployment units*)



A simple example: a trivial prototype of a “banking application” - Processes and threads (*runtime units*)



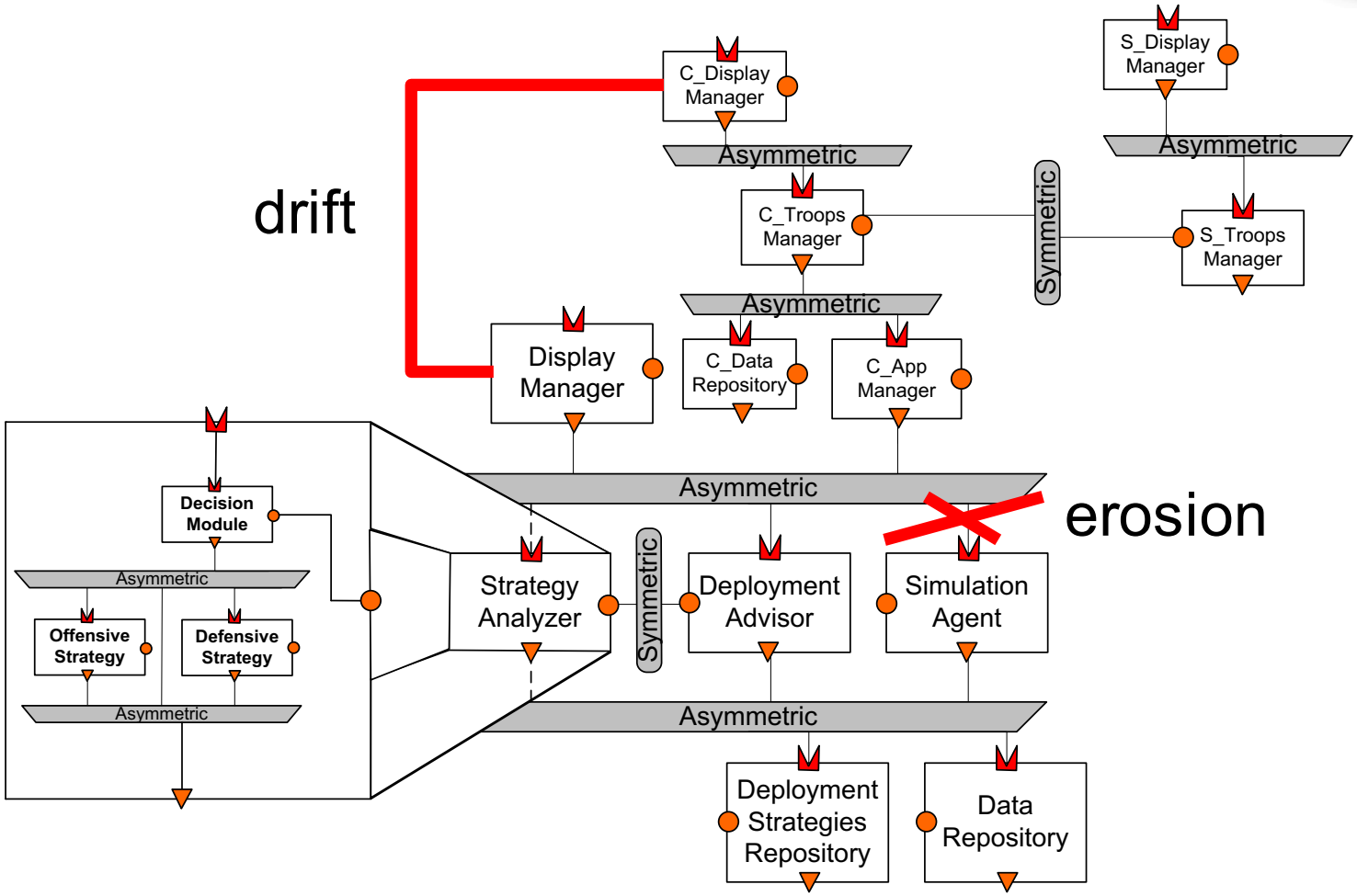
What happens if changes are applied on the source code?

- Architectural degradation
  - Changes in the software system are not reflected in the corresponding architectural description
- Why does this happen?
  - Lack of a documented architecture
  - Inadequate processes and supporting tools
  - Developer sloppiness
  - Perception of short deadlines

Drift and erosion

- Two kinds of degradation
  - **Drift:** the change is not included nor implied by the architecture but it does not necessarily imply its outright violation
  - **Erosion**: the change violates the described architecture

Drift and erosion, an example

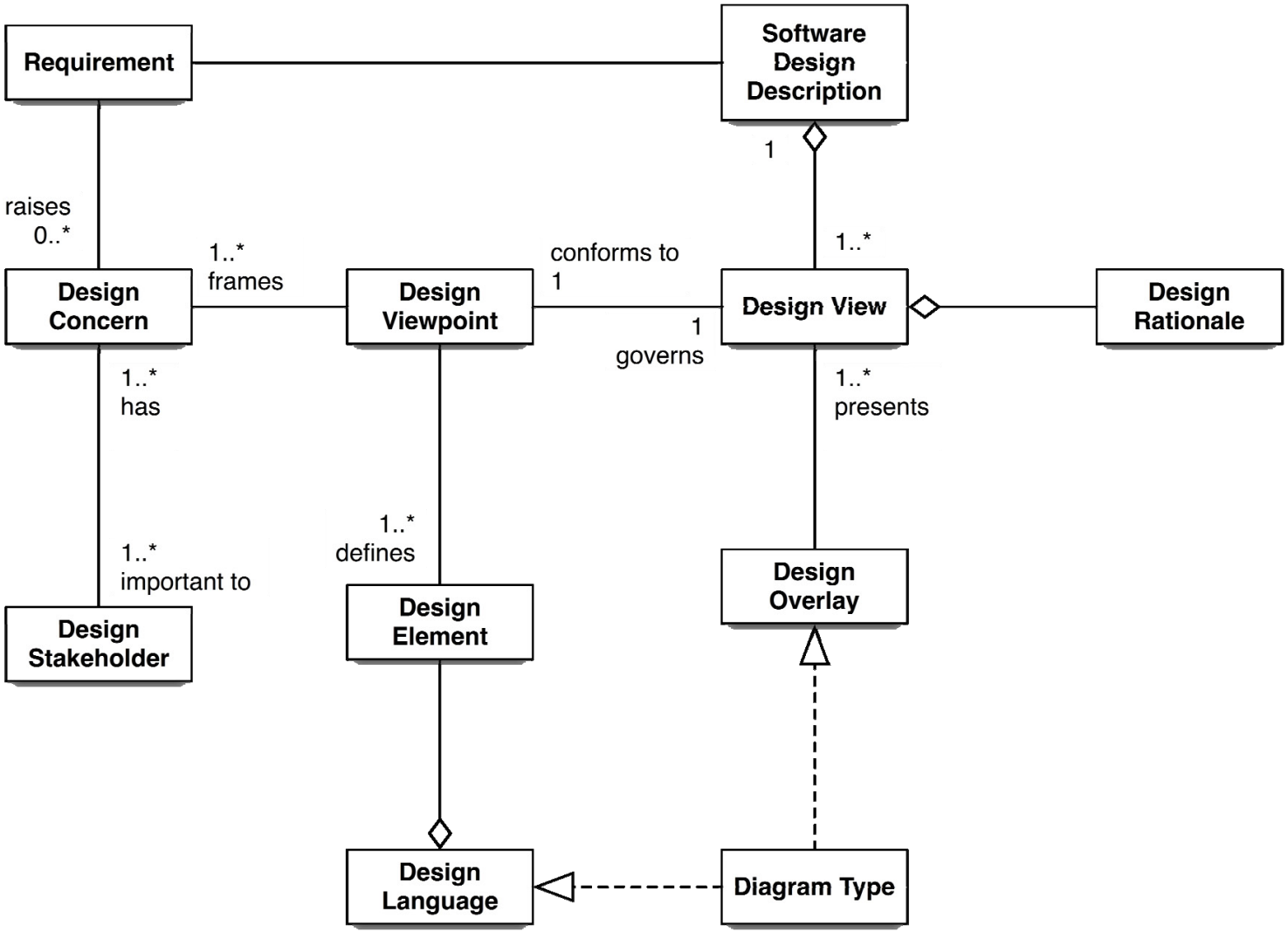


Software Design Description

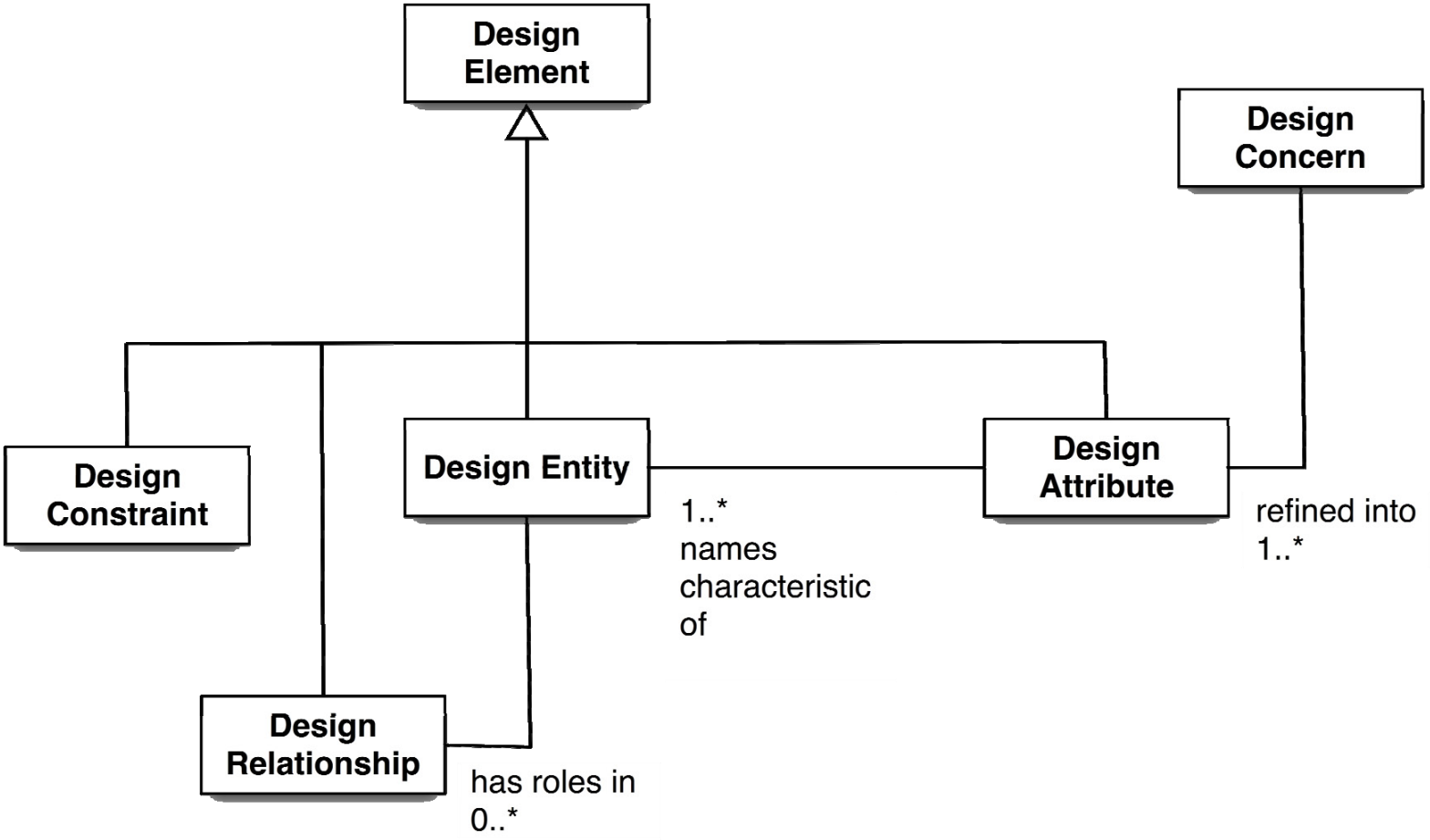
Software Design Description (SDD)

- The IEEE Standard for Information Technology—Systems Design—Software Design Descriptions
  - "Describes software designs and establishes the information content and organization of a software design description."
  - "An SDD is a representation of a software design to be used for recording design information and communicating that design information to key design stakeholders."
- The IEEE Standard for Systems and software engineering — Architecture description
  - Is aligned with the previous one and
  - Specifies "the manner in which architectural descriptions of systems are organized and expressed"

The key concepts of a SDD according to the IEEE standard (1)



The key concepts of a SDD according to the IEEE standard (2)



The required contents of an SDD according to the IEEE standard

- Identification of the SDD
- Identified design stakeholders
- Identified design concerns
- Selected design viewpoints, each with type definitions of its allowed design elements and design languages
- Design views
- Design overlays
- Design rationale

Software Architecture & Quality

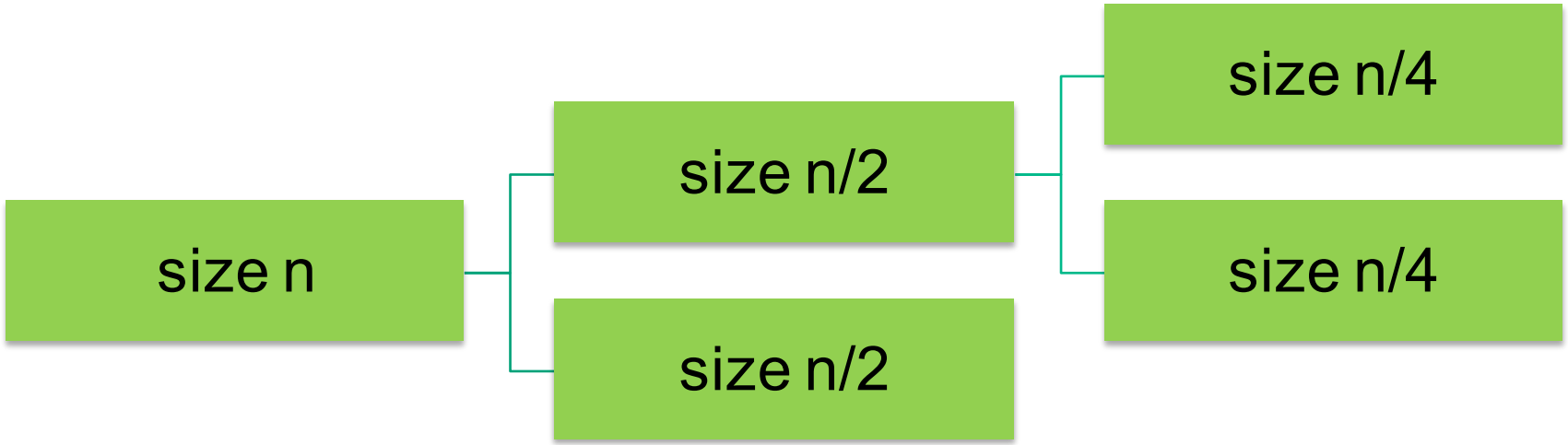
- The notion of quality is central in software architecting: a software architecture is devised to gain insight in the qualities of a system at the earliest possible stage.
- Some qualities are observable via execution: performance, security, availability, functionality, usability
- And some are not observable via execution: modifiability, portability, reusability, integrability, testability

Design principles

Design Principles

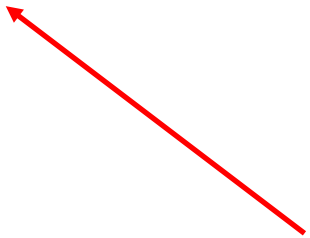
- Design Principle 1: Divide and conquer
- Design Principle 2: Increase cohesion where possible
- Design Principle 3: Reduce coupling where possible
- Design Principle 4: Keep the level of abstraction as high as possible
- Design Principle 5: Increase reusability where possible
- Design Principle 6: Reuse existing designs and code where possible
- Design Principle 7: Design for flexibility
- Design Principle 8: Anticipate obsolescence
- Design Principle 9: Design for portability
- Design Principle 10: Design for testability
- Design Principle 11: Design defensively

Design Principle 1: Divide & Conquer: the binary search example



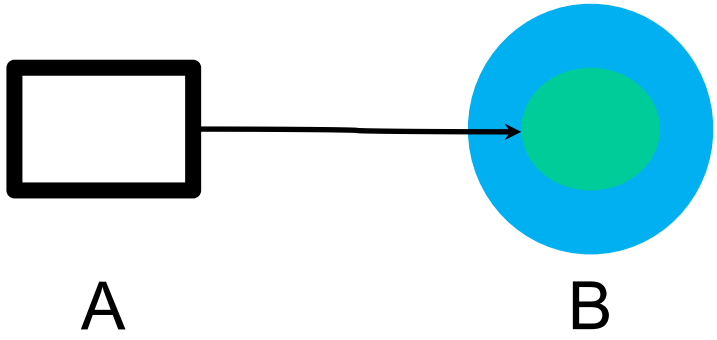
Design Principle 2: Cohesion

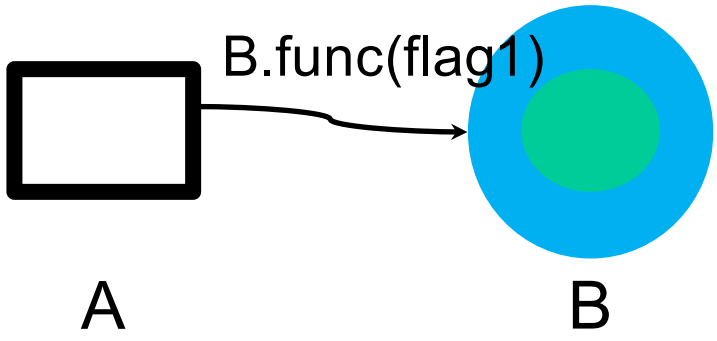
- Example of a non-cohesive module
  - Class Utility {
    - ComputeAverageScore(Student s[])
    - ReduceImage(Image i)
  - }

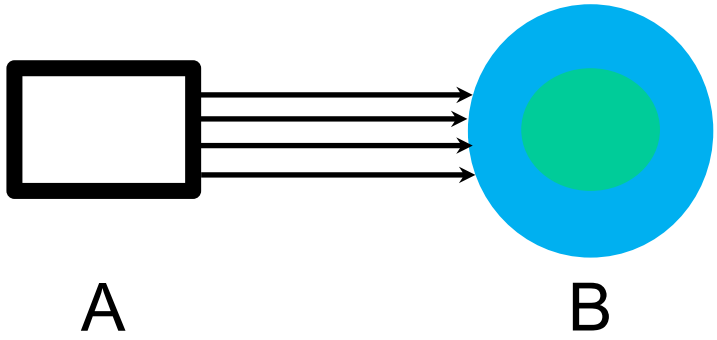


Is this a cohesive class???

Design Principle 3: Coupling
What to avoid







A is accessing the
data structure in B
breaking encapsulation

```
class b {
  func(flag f) {
    if(f == flag1) do this
    else if (f == flag2) do that
    else…
  }
}
```

Do we really need so many
messages from A to B?

Design Principle 4: Keep the level of abstraction as high as possible

- 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

Design Principle 5: Increase reusability where possible

- Design the various aspects of your system so that they can be used again in other contexts
  - Generalize your design as much as possible
  - Follow the preceding three design principles
  - Design your system to contain hooks
  - Simplify your design as much as possible

Design Principle 6: Reuse existing designs and code where possible

- Design with reuse is complementary to design for reusability
  - Actively reusing designs or code allows you to take advantage of the investment you or others have made in reusable components
    - *Cloning* should not be seen as a form of reuse

Design Principle 7: Design for flexibility

- Actively anticipate changes that a design may have to undergo in the future, and prepare for them
  - Reduce coupling and increase cohesion
  - Create abstractions
  - Do not hard-code anything
  - Leave all options open
    - Do not restrict the options of people who have to modify the system later
  - Use reusable code and make code reusable

Design Principle 8: Anticipate obsolescence

- Plan for changes in the technology or environment so the software will continue to run or can be easily changed
  - Avoid using early releases of technology
  - Avoid using software libraries that are specific to particular environments
  - Avoid using undocumented features or little-used features of software libraries
  - Avoid using software or special hardware from companies that are less likely to provide long-term support
  - Use standard languages and technologies that are supported by multiple vendors

Design Principle 9: Design for Portability

- Have the software run on as many platforms as possible
  - Avoid the use of facilities that are specific to one particular environment
  - E.g. a library only available in Microsoft Windows

Design Principle 10: Design for Testability

- Take steps to make testing easier
  - Design a program to automatically test the software
    - Ensure that all the functionality of the code can by driven by an external program, bypassing a graphical user interface
  - In Java, you can
    - create a main() method in each class in order to exercise the other methods
    - Use Junit and related approaches to build an automated testing framework for your system

Design Principle 11: Design defensively

- Never trust how others will try to use a component you are designing
  - Handle all cases where other code might attempt to use your component inappropriately
  - Check that all of the inputs to your component are valid: the *preconditions*
    - Unfortunately, over-zealous defensive design can result in unnecessarily repetitive checking