Advanced Topics of Software Engineering (ASE)

Chapter 2. From requirements to system design

Prof. Dr. Florian Matthes, Prof. Dr. Alexander Pretschner

Chair of Software Engineering for Business Information Systems (sebis)
Faculty of Informatics
Technische Universität München
wwwmatthes.in.tum.de

From requirements to system design

Formal definition of the structure of a system (1)

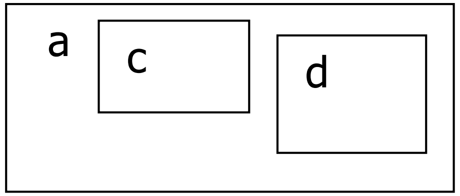
- Let us first formally define the structure of **system S**
  - by the tuple $S \coloneqq (C, I, CON)$
  - with C denoting the **components**
  - with $env \in C$ denoting the system **environment**
  - with I denoting the **interfaces** of the components
  - and $CON \subseteq I \times I$ denoting the **connection between interfaces**.

- A component can be a system itself
  - this establishes a hierarchy of components
  - Example see next slide

Formal definition of the structure of a system (2)

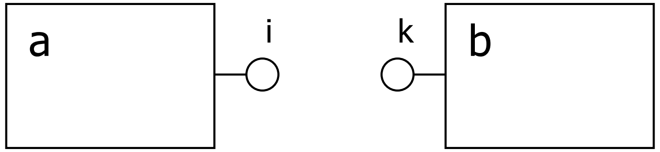
- Parent relationship between components (total function)
  - Example
    - parent(c) = parent(d) = a
    - parent(a) = *env*

$$\boldsymbol{parent}: C \rightarrow C$$



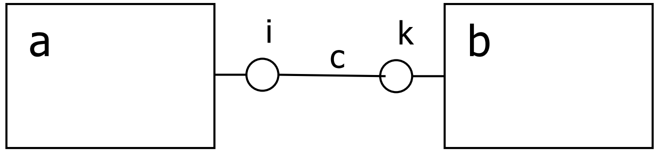
- Interface-component relationship (total function)
  - Example
    - assigned(i) = a
    - assigned(k) = b

$$\boldsymbol{assigned}: I \rightarrow C$$

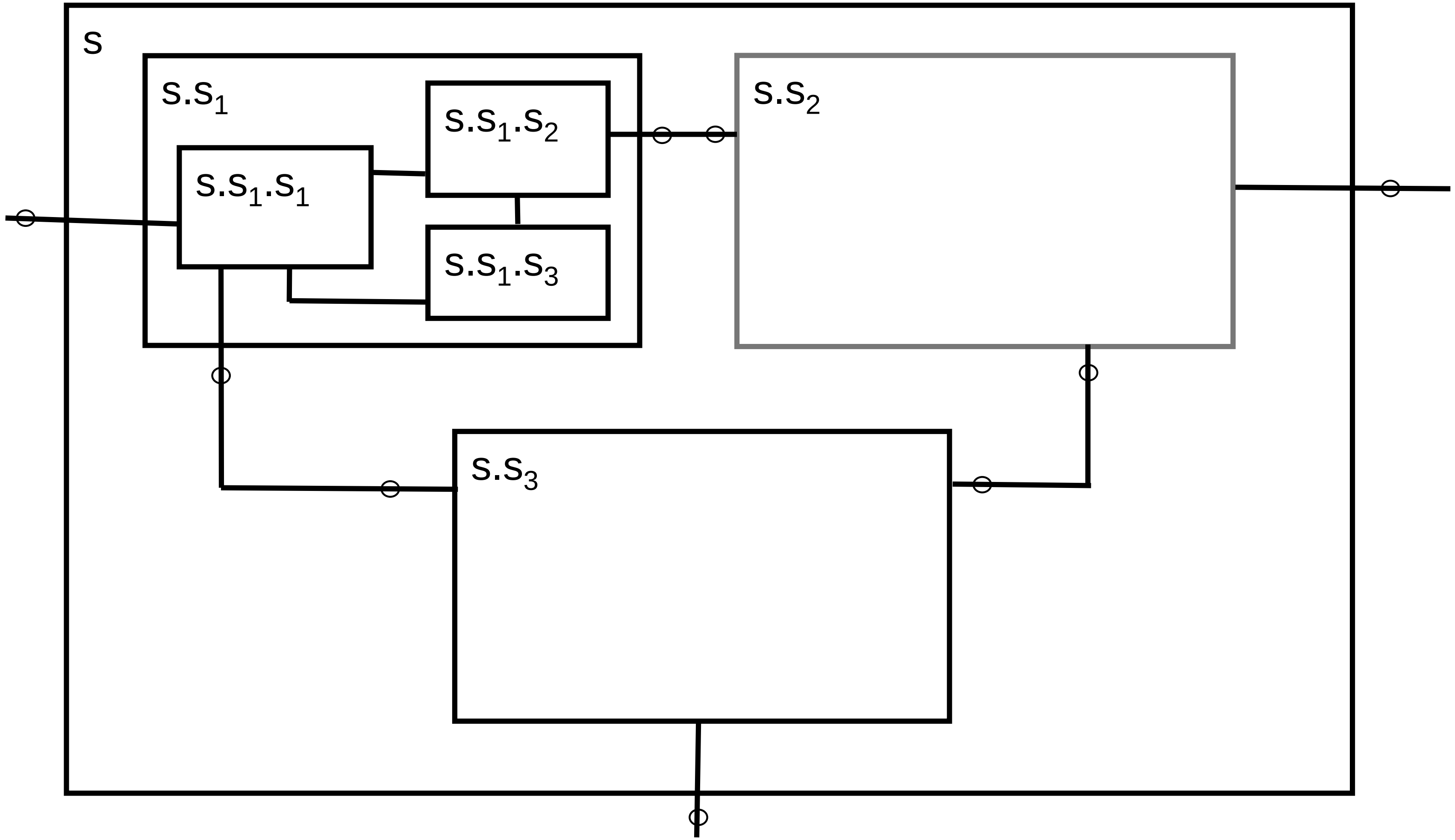


- Connection (CON) between interfaces (total function)
  - Example
    - connected(c)=(i,k)

$$\boldsymbol{connected}: CON \rightarrow (I \times I)$$



Hierarchical structure of a system



Quantitatively measuring coupling

- For the sake of simplicity let us consider coupling of a system as the *normalized number of connections between components at the same hierarchical level ("not so bad", factor* α*) and the normalized number of connections at different hierarchical levels ("a bit worse", factor 1-* α*).*

$$coupling: S \rightarrow \mathbb{R}$$

$$coupling(s) := \alpha * \frac{\left|\left\{con \in s.CON \,\middle|\, \exists i,j \in s.I: con = connected(i,j) \wedge parent\left(assigned(i)\right) = parent\left(assigned(j)\right)\right\}\right|}{|s.C|}$$

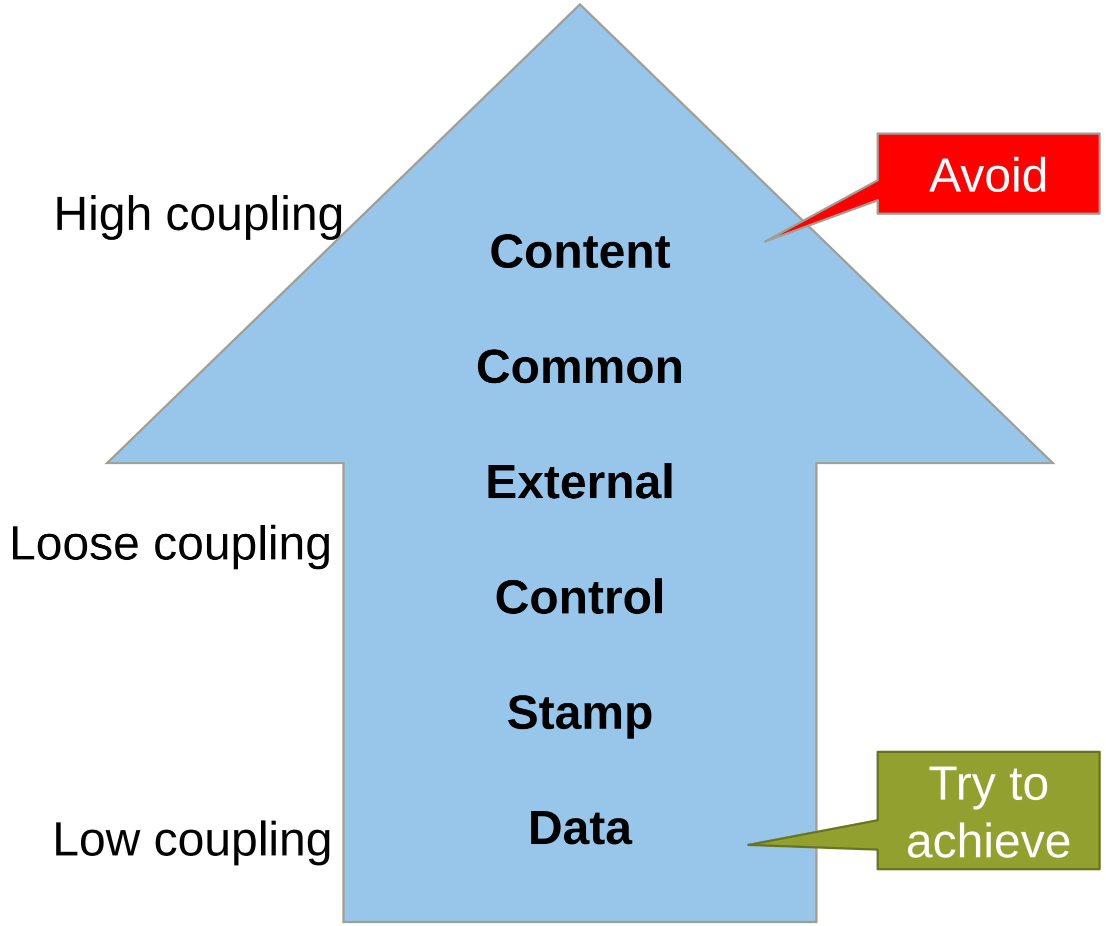
$$+ (1-\alpha) * \frac{\left|\left\{con \in s.CON \,\middle|\, \exists i,j \in s.I: con = connected(i,j) \wedge parent\left(assigned(i)\right) \neq parent\left(assigned(j)\right)\right\}\right|}{|s.C|}$$

- Note: One could put more emphasis on the top-level components than on the lower-level components.

- The coupling of a system s_1 is smaller than that of s_2
  - iff $coupling(s_1) < coupling(s_2)$

For a different treatment and more details, see ["A unified framework for coupling measurement in object-oriented systems." Briand, L. C., Daly, J. W., and Wüst, J. K. (1999)]

Types of coupling in software



[Content, Common, etc. are interaction patterns, described next]

["Software Engineering: Principles and Practice." Van Vliet H. (2008)]

Content coupling

- One component directly affects the working of another component.
- Content coupling occurs when a component changes another component's data or when control is passed from one component to the middle of another
  - (as in a jump).

- Example - component A handles 'customer lookup'
  - When a customer record is not found, component A adds the customer by directly modifying the content of the data structure containing customer information (which is the responsibility of component B - 'creating customers’).

Almost any change to component B requires changes to component A.

["Software Engineering: Principles and Practice." Van Vliet H. (2008)]

Common coupling

- Two components have shared data.

- The name originates from the use of COMMON blocks in FORTRAN.

- Its equivalent in block-structured languages is the use of global variables.

- Lack of clear responsibility for the data
- Reduces readability
- Difficult to determine all the components that affect a data element (reduces maintainability)
- Difficult to reuse components
- Reduces ability to control data accesses

["Software Engineering: Principles and Practice." Van Vliet H. (2008)]

External coupling

- Components communicate through an external medium such as a:
  - File
  - Device interface
  - Protocol
  - Data format

["Software Engineering: Principles and Practice." Van Vliet H. (2008)]

Control coupling

- One component directs the execution of another component by passing the necessary control information.
  - This is usually accomplished by means of flags that are set by one component and reacted upon by the dependent component.

- May be either *good or bad*, depending on the situation.
  - *Good* if parameters allow factoring and reuse of functionality.
  - Example - sort that takes a comparison function as an argument.
    - The sort function is clearly defined: return a list in sorted order, where sorted is determined by a parameter.

  - *Bad* if parameters indicate completely different behavior or
  - if components are not independent
    - Component B must know the internal structure of component A – might affect reusability

["Software Engineering: Principles and Practice." Van Vliet H. (2008)]

Stamp coupling

- Complete data structures are passed from one component to another.
- With stamp coupling, the precise format of the data structures is a common property of those components

- Example
  - calculateSalary(Employee employee)

The second component has more information than it needs

Define interfaces to limit access from clients

["Software Engineering: Principles and Practice." Van Vliet H. (2008)]

Data coupling

- Component passes data (not data structures) to another component
- Good, if it can be achieved
- Example
  - calculateSalary(String name, int noOfHours, int salPerHour)

["Software Engineering: Principles and Practice." Van Vliet H. (2008)]

Cohesion

- How closely related are the different responsibilities of a component
- The degree of interaction within a component
- Make cohesion as strong as possible

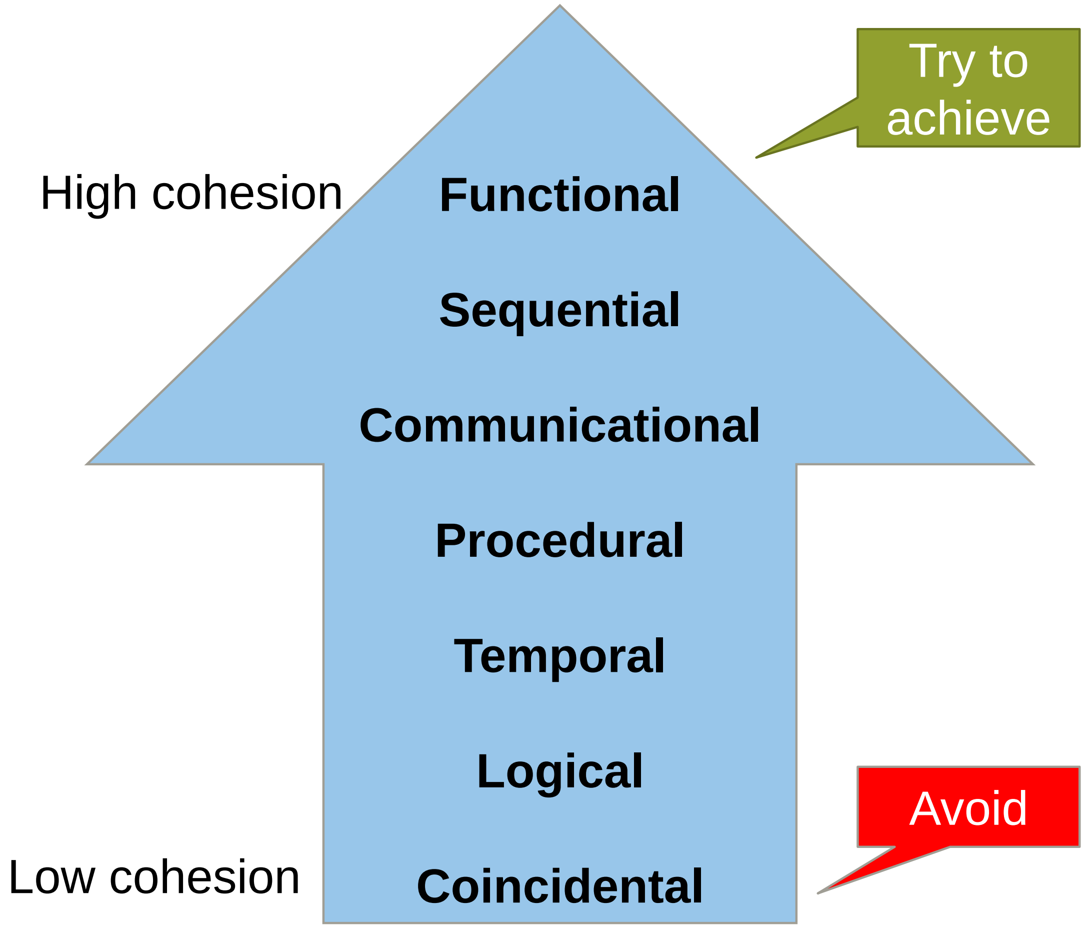
Coupling and cohesion are highly interrelated

A component's cohesion characterizes its internal interdependencies.

Strive for low coupling and high cohesion

["Software Engineering: Principles and Practice." Van Vliet H. (2008)]

Types of cohesion



["Structured Design" Yourdon, E. and Constantine, L. L. (1975)]

Coincidental cohesion

- Elements are grouped into components in a haphazard way.
- There is no significant relation between the elements.
- Parts of the component are unrelated (unrelated functions, processes, or data)
- Accidental
- Worst form

- Example – a component that
  - Prints next line
  - Reverses string of characters in 2nd argument
  - Adds 7 to 3rd argument
  - Converts 4th argument to float

Degrades maintainability and components are not reusable

Break into separate modules each performing one task

["Software Engineering: Principles and Practice." Van Vliet H. (2008)]

Logical cohesion

- Elements of component are related logically and not functionally.
- Several logically related elements are in the same component and one of the elements is selected by the client component.

- Example:
  - A component reads inputs from tape, disk, and network.
  - All the code for these functions are in the same component.
  - Operations are related, but the functions are significantly different.

["Software Engineering: Principles and Practice." Van Vliet H. (2008)]

Temporal cohesion

- The elements are independent, but they are activated at about the same point in time.

- Initialization component example
  - open old db, new db, transaction db, print db
  - initialize sales district table
  - read first transaction record
  - read first old db record

- Actions weakly related to one another, but strongly related to actions in other modules
- Code spread out --- not maintainable or reusable

For the above example, define these initializers in the proper modules and then have an initialization module call each operation.

["Software Engineering: Principles and Practice." Van Vliet H. (2008)]

Procedural cohesion

- Elements of a component are related only to ensure a particular order of execution.

- Example
  - Write output record
  - Read new input record
  - Pad input with spaces
  - Return new record

Actions are still weakly connected and unlikely to be reusable.

["Software Engineering: Principles and Practice." Van Vliet H. (2008)]

Communicational cohesion

- The elements of a component operate on the same (external) data.

- Example:
  - Update the record in a database
  - Print the record

Still leads to less reusability --- break it up

["Software Engineering: Principles and Practice." Van Vliet H. (2008)]

Sequential cohesion

- The output of one part is the input to another.
- Data flows between parts (different from procedural cohesion).
- Occurs naturally in functional programming languages.
- Good situation

["Software Engineering: Principles and Practice." Van Vliet H. (2008)]

Functional cohesion

- Every essential element to a single computation is contained in the component
- Such a component often transforms a single input into a single output
- Ideal situation

- Increases
  - Reusability
  - Testability
  - Understandability
  - Learnability
- Corrective maintenance is easier
  - Fault isolation
  - Reduced regression faults
- Easier to extend product (extensibility)

["Software Engineering: Principles and Practice." Van Vliet H. (2008)]

Conclusion

Strive for low coupling and high cohesion

- Communication between programmers becomes simpler.
  - Decisions can be made locally and without interfering with other components.
- Correctness proofs become easier to derive.
- It is less likely that changes will propagate to other components.
  - Avoids ripple-effects (low coupling)
  - Allows changes to be local (high cohesion)
  - Reduces maintenance costs
- Increases reusability
- The comprehensibility of components is increased.
  - Manifests in small set of component interfaces
  - Simple component interfaces allow for an understanding of a component independent of the context in which it is used
- Less error-prone
- But … don’t overdo it!

["Software Engineering: Principles and Practice." Van Vliet H. (2008)]

From requirements to system design

[… tbc]

Single responsibility principle

"There should never be more than one reason for a class to change."

["Agile Software Development, Principles, Patterns, and Practices." Martin, R. C.]

- A class should only have one reason to change.

- If there are two reasons for a class to change, we have to split the functionality into two classes.
  - Each class will handle only one responsibility.
  - If we need to make a change, we should make it in the class that handles it.

- This leads to cohesion at the package/implementation level.

- The single responsibility principle is a simple and intuitive principle, but in practice it is sometimes hard to get it right!

Responsibility

"… a responsibility is a family of functions that serves one particular actor."

["Agile Software Development, Principles, Patterns, and Practices." Martin, R. C.]

- Core functionality of a class (what a class does)
- The more a class does, the more likely it will change.
- The more a class changes, the more likely it will affect its associated classes.

Single responsibility principle

["SOLID development principles." Bailey D. (2009)]

From requirements to system design

2.1. Software architecture

- 2.1.1. Software modules and software components
- 2.1.2. Dependency structure matrix
- 2.1.3. Guidelines for modular design
  - 2.1.3.1. Low coupling and high cohesion
  - 2.1.3.2. Single responsibility principle
  - **2.1.3.3. Separation of concerns**
  - 2.1.3.4. Liskov’s substitution principle
  - 2.1.3.5. Interface-segregation principle
  - 2.1.3.6. Anticipate change
  - 2.1.3.7. Don't repeat yourself
- 2.1.4 Architecture and external quality

2.2. Antipatterns

[… tbc]

Separation of concerns

- Talk about different things ("features") in different places
  - Functional and non-functional
  - Example: business logic, presentation, data layer

- In functional context, related to the principle of low coupling and high (functional) cohesion.

- Addresses limitations of human cognition.

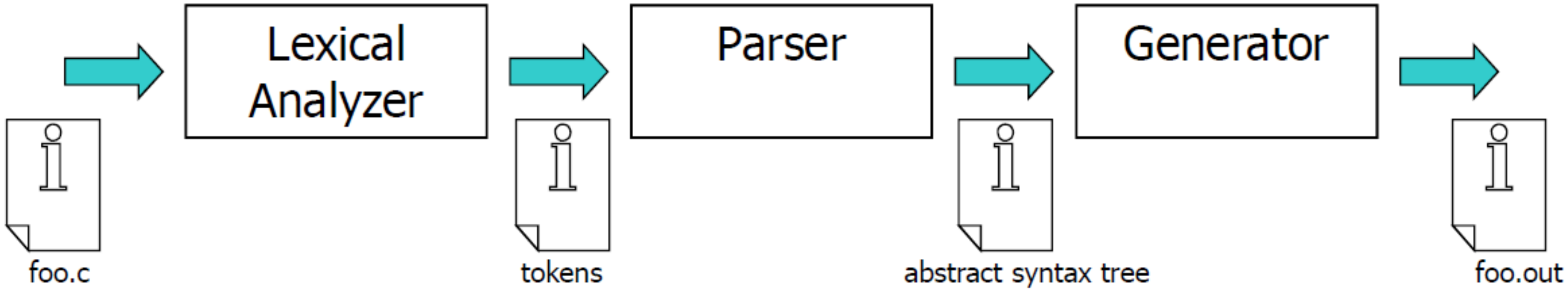
- Humans are only able to process ~7 data units at a time.

- Thus, functionality should not be overly scattered.

- Minimize responsibilities-per-component ratio.

- Components should only encapsulate semantically related functionalities.

Separation of concerns



- Each component fulfills its own distinct purpose.
- Functionalities are encapsulated within components.
- Components can be replaced arbitrarily.

Open/closed principle

Software entities (classes, modules and functions) should be open for extension, but closed for modifications.

["Object-Oriented Software Construction." Bertrand Meyer (1988)]

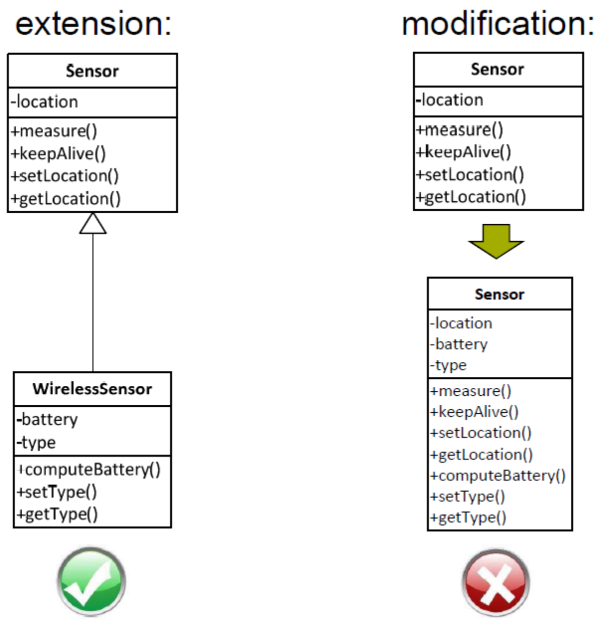
- A module will be said to be ***open*** if it is still available for extension.
  - For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.

- A module will be said to be ***closed*** if it is available for use by other modules.
  - This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding).

["Agile Software Development, Principles, Patterns, and Practices." Martin R. C. (2002)]

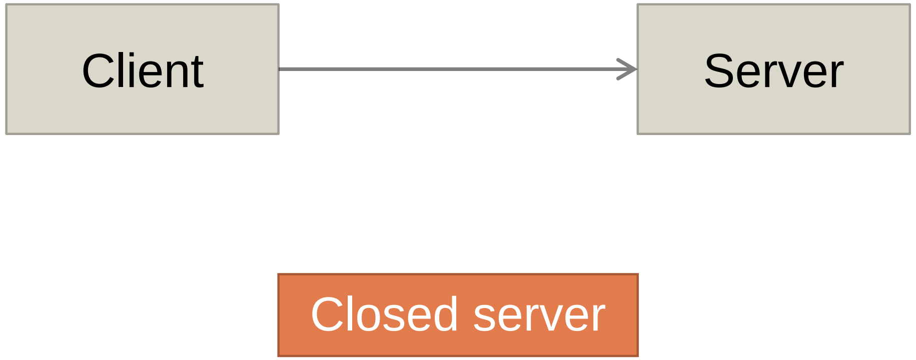
Conforming to the open/closed principle

- Open for extension
  - Behavior of the module can be extended
  - We are able to change what the module does
- Closed for modification
  - Extending behavior does not result in changes to the source, binary, or code of the module
  - Avoids unanticipated effects on dependent components
- Highly related to the inheritance and polymorphism paradigm of object-oriented programming



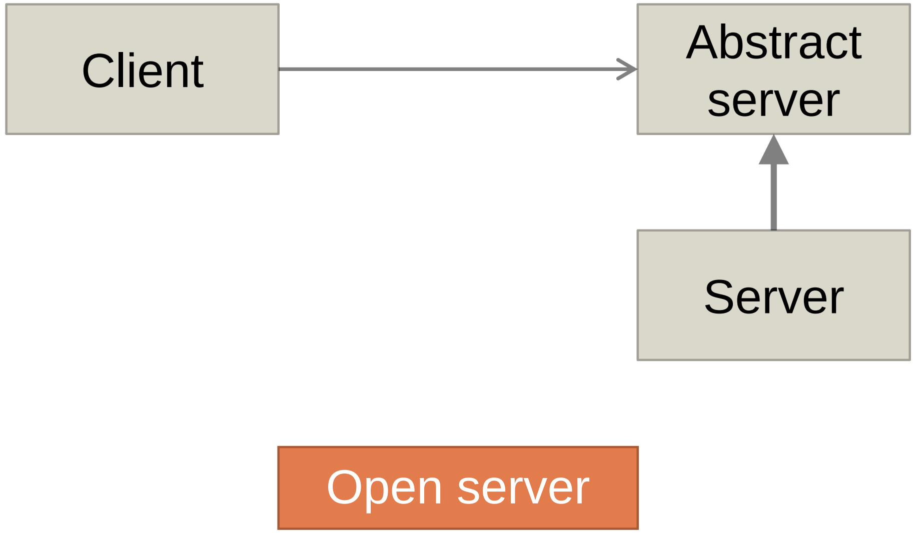
["Agile Software Development, Principles, Patterns, and Practices." Martin R. C. (2002)]

Does not conform to the open/closed principle



["Agile Software Development, Principles, Patterns, and Practices." Martin R. C. (2002)]

Strategy pattern



- Conforming to the open/closed principle
- Relies on abstractions
  - Interfaces
  - Abstract classes

["Agile Software Development, Principles, Patterns, and Practices." Martin R. C. (2002)]

From requirements to system design

2.1. Software architecture

- 2.1.1. Software modules and software components
- 2.1.2. Dependency structure matrix
- 2.1.3. Guidelines for modular design
  - 2.1.3.1. Low coupling and high cohesion
  - 2.1.3.2. Single responsibility principle
  - 2.1.3.3. Separation of concerns
  - **2.1.3.4. Liskov’s substitution principle**
  - 2.1.3.5. Interface-segregation principle
  - 2.1.3.6. Anticipate change
  - 2.1.3.7. Don't repeat yourself
- 2.1.4 Architecture and external quality

2.2. Antipatterns

[… tbc]

Liskov’s substitution principle (LSP)

"Let *q(x)* be a property provable about objects *x* of type *T*. Then *q(y)* should be provable for objects *y* of type *S* where *S* is a subtype of *T*."

[Barbara Liskov (1987)]

"Subtypes must be substitutable for their base types"

["Agile Software Development, Principles, Patterns, and Practices." Martin R C. (2002)]

- Child classes should never break the parent class' type definitions.
- All derived classes must be substitutable for their base classes.

- Substitutability
  - Child classes must **not:**
    - Remove base class behavior
    - Strengthen preconditions nor weaken post-conditions
    - Violate base class invariants
    - And in general, must not require calling code to know they are different from their base type
- This principle guides us in the creation of abstractions

Example: Violation of Liskov's Substitution Principle

```
class Rectangle {
  protected int m_width;
  protected int m_height;
  public void setWidth (int width) { m_width = width; }
  public void setHeight (int height) { m_height = height; }
  public int getWidth() { return m_width; }
  public int getHeight() { return m_height; }
  public int getArea() { return m_width * m_height; }
}
```

Postcondition: m_height does not change

Postcondition: m_width does not change

```
class Square extends Rectangle {
  public void setWidth (int width) {
      m_width = width;
      m_height = width;
  }
  public void setHeight (int height) {
      m_width = height;
      m_height = height;
  }
}
```

Violation of postcondition in super class!

Example: Violation of Liskov's Substitution Principle

```
class Rectangle {
  protected int m_width;
  protected int m_height;
  public void setWidth (int width) { m_width = width; }
  public void setHeight (int height) { m_height = height; }
  public int getWidth() { return m_width; }
  public int getHeight() { return m_height; }
  public int getArea() { return m_width * m_height; }
}
```

Postcondition: m_height does not change

Postcondition: m_width does not change

```
class Square extends Rectangle {
  public void setWidth (int width) {
     m_width = width;
     m_height = width;
  }
  public void setHeight (int height) {
     m_width = height;
     m_height = height;
  }
}
```

```
class LspTest {
   private static Rectangle getNewRectangle() {
      return new Square();
   }

   public static void main (String args[]) {
      Rectangle r = LspTest.getNewRectangle();
      r.setWidth(5);
      r.setHeight(10);
      System.out.println(r.getArea());
   }
}
```

The area is 100 instead of 50!

The inheritance rules of OO languages are not sufficient

- OO inheritance relates to syntax only, not to semantics (pre- and post-conditions, invariants).
- The semantics of classes are not necessarily syntactic only. We may add behavior, e.g., in the form of pre- and post-conditions, or class invariants that hold before and after executing a method. Remember the discussion on semantic interfaces.
- Then we may require a subclass to **weaken preconditions** (not just contravariant argument types), to **strengthen postconditions** (not just covariant return types) and **establish invariants** that don't break the super class invariant. Liskov called this behavioral subtyping.
- Liskov's principle captures this idea (read the slide again!)

From requirements to system design

[… tbc]

Interface-segregation principle (ISP)

Clients should not be forced to depend upon interfaces they don't use.

["Agile Software Development, Principles, Patterns, and Practices." Martin R C. (2002)]

- Related to single responsibility principle
- Prefer small, cohesive interfaces to "fat" interfaces

Violation of ISP

```
interface IEmployee {
  public int calculateWorkHrs();
  public int getNoOfVacations();
}
```

```
class WiMi implements IEmployee {
  public int calculateWorkHrs() {
    // .... returns the work hrs
  }
  public int getNoOfVacations() {
    // ...... returns no of days in vacation sheet
  }
}
```

```
class HiWi implements IEmployee {
  public int calculateWorkHrs() {
    //.... returns the work hrs
  }
  public int getNoOfVacations() {
    throw new NotImplementedException();
  }
}
```

```
class Secretary {

  public List getAllEmployees() {
    // returns a list of hiwis and wimis
  }

  public void manageVacations() {
    List allEmployees = getAllEmployees();
    for(IEmployee emp : allEmployees) {
      int nov = emp. getNoOfVacations();
    }
  }
}
```

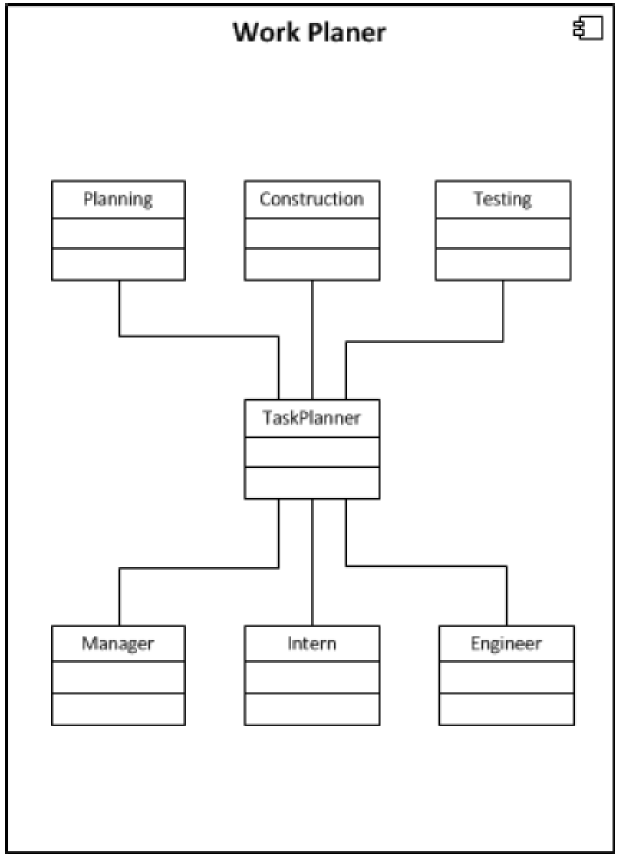
Flughafen gate, passanger repository, Boarding interface, managmeent interface, Passenger mgmt system; 2 folien good vs bad

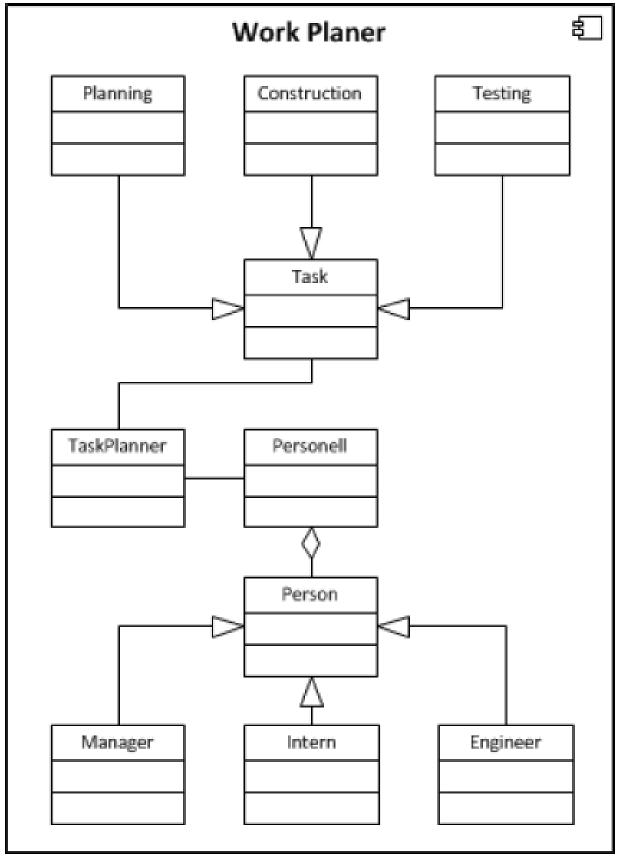
From requirements to system design

Anticipate change

- Build components in a way that minimizes effort for potential future changes.
- Find compromise between generality and specificity.
- Interface definition should consider potential extensions and changes.
- Low coupling and high cohesion ease changes.
- Don’t overdo it, though. Managed reuse is terribly difficult (and often terribly unsuccessful) – we’ll talk about product lines later.

Anticipate change





From requirements to system design

[… tbc]

Don't repeat yourself (DRY)

"Every piece of knowledge (and functionality) must have a single, unambiguous, authoritative representation within a system."

- Programmers are constantly in maintenance mode – the understanding of the system changes day by day.
- To perform maintenance, we have to change the representation of things.
- If you have more than one way to express the same thing, at some point the two or three different representations will fall out of step with each other.
- Redundant modules and interfaces that serve the same or almost the same purpose are problematic in terms of avoidable system complexity and ambiguity.
- Unnecessary repetition increases the risk of inconsistencies, increases maintenance effort, and reduces general understandability.

["The pragmatic programmer: from journeyman to master." Hunt A. and Thomas D. (2000)]

How does duplication arise?

- **Imposed** duplication
  - Developers feel they have no choice – the environment seems to require duplication.
- **Inadvertent** duplication
  - Developers don’t realize that they are duplicating information.
- **Impatient** duplication
  - Developers get lazy and duplicate because it seems easier.
- **Inter-developer** duplication
  - Multiple people on a team (on different teams) duplicate a piece of information.

Imposed duplication

- Multiple representations of information, e.g.:
  - Client-server application, using different languages on the client and server, both need to represent some shared structure on both.
  - Classes which mirror the schema of a database table.
  - A book which includes code samples which have to be compiled and tested.
  - Documentation in code
    - Keep low-level knowledge in the code, reserve the comments for other, high-level explanations.
  - Language issues: Many languages separate a module’s interface from its implementation, e.g.:
    - C and C++ header files, CORBA-IDL (the compiler helps here)

Have a single authoritative representation that then generates non-authoritative work products, like code or DDLs (data description languages).

Inadvertent duplication (1)

Bad design decisions

- Logistics example:
  - Trucks have a type, a driver, a license number
  - Delivery routes contain a driver, a truck, a route
  - If we encode the driver twice, then we need to change it twice, e.g., if a driver calls in sick.

Inadvertent duplication (2)

Comes as a result of mistakes in the design

```
class Line {
  Point start;
  Point end;
  double length;
}
```

Need to re-compute whenever start or end change

```
class Line {
  Point start;
  Point end;
  double length() {
    return start.distanceTo(end);
  }
}
```

No redundancy for length any more

```
class Line {
  private Point start;
  private Point end;
  private double length;
  private boolean changed = true;

  public void setStart(Point p) {
    start = p; changed = true;
  }
  public void setEnd(Point p) {
    end = p; changed = true;
  }
  public Point getStart() {
    return start;
  }
  public Point getEnd() {
    return end;
  }
  double getLength() {
    if(changed) {
      length = start.distanceTo(end);
      changed = false;
    }
    return length;
  }
}
```

But … caching for complex operations may be adequate

Impatient and inter-developer duplication

Impatient duplication arises due to:

- Time pressure → shortcuts
- It takes discipline and a willingness to spend time upfront to save pain later.

Inter-developer duplication can be avoided:

- Have a clear design, a strong technical project leader, and a well-understood division of responsibilities within the design.
- Encourage communication between developers.
- Have a central place in the source tree where utility routines and scripts can be deposited.
- Make a point of reading other people's source code and documentation (informally or during code reviews).
- Apply common sense!

Make it easy to reuse (reusability will be covered later in this lecture)
… oh, and don't overdo it!

From requirements to system design

[… tbc]

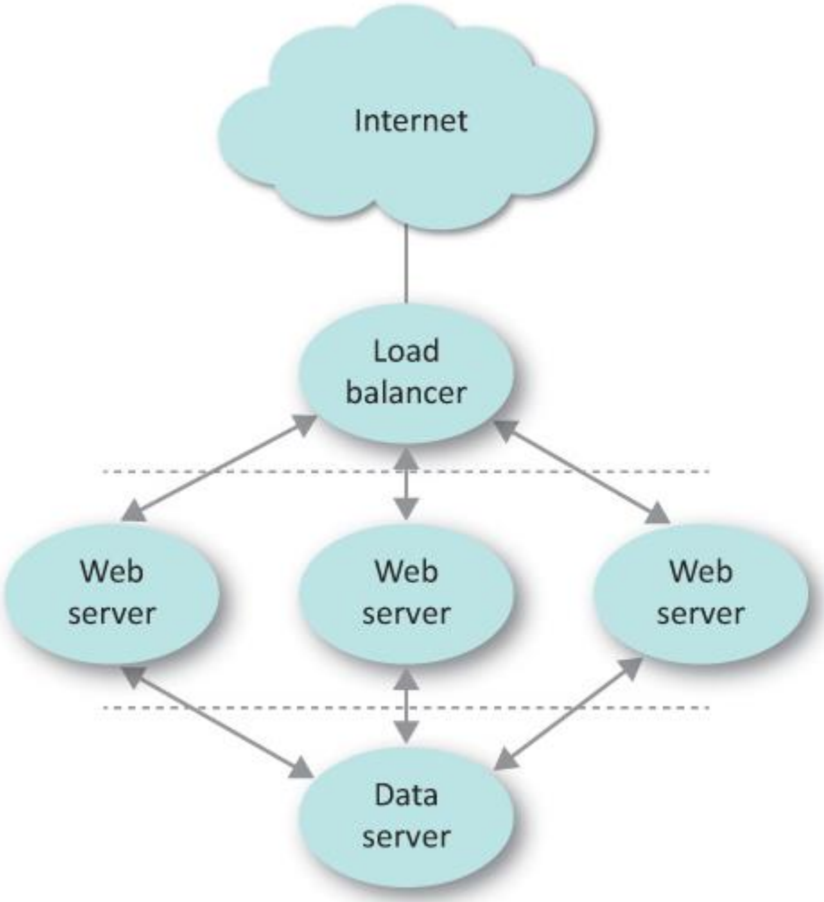
Back to the big picture

- Internal quality:
  - Modularity (positively) influences maintainability and reusability.
  - It also (both positively and negatively) influences testability – we will discuss this later.

- External quality:
  - We have seen that many layers may lead to many indirections, resulting in bad performance.
  - We have seen that a large attack surface makes a system vulnerable – architecture impacts security.

Architecture and security (1)

3- and 4-tier architectures



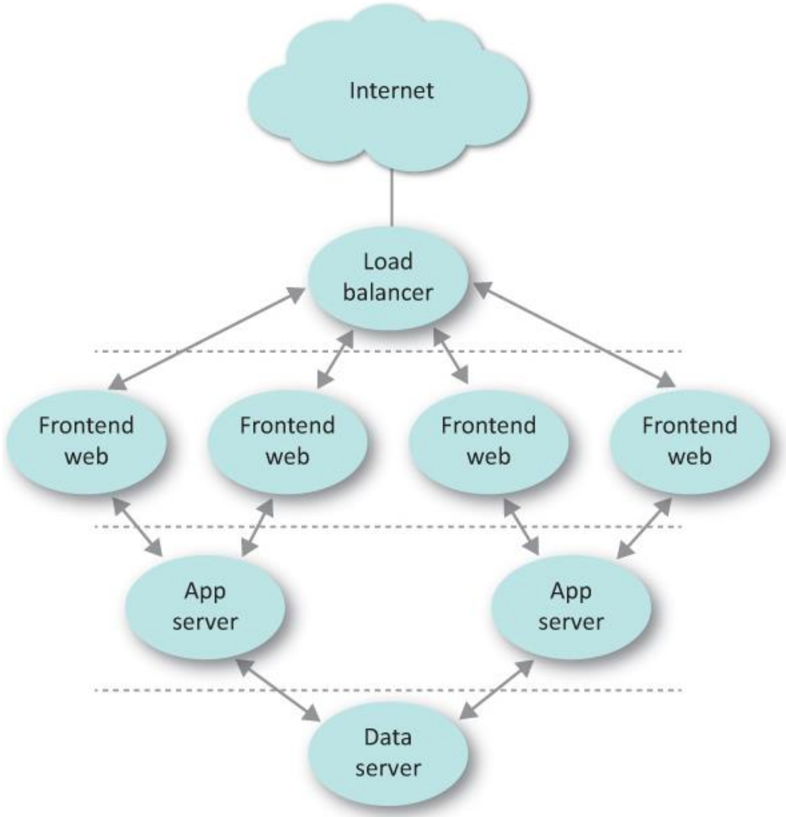
Limoncelli, T; Chalup, S.; Hogan, C.: **The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems, Volume 2**
Addison-Wesley Professional, 2014

- 3 tiers
- Web server also runs application server
- Compromising web server facilitates (or is identical to) compromise of applications

+ efficient
+ "all in one hand"
- Rather large attack surface for applications

Architecture and security (2)

3- and 4-tier architectures



Limoncelli, T; Chalup, S.; Hogan, C.: **The Practice of Cloud System Administration: Designing and Operating Large Distributed Systems, Volume 2**
Addison-Wesley Professional, 2014

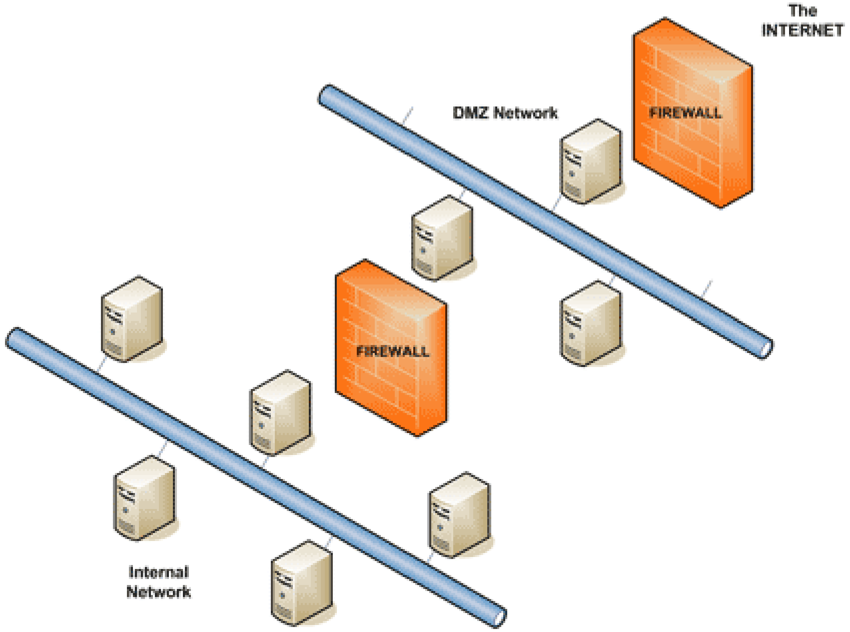
- 4 tiers
- Web and application servers separated
- Compromised web server only the first step to also compromise the applications

+ attack surface reduced

- Operations of web server and app server possibly in different hands

Architecture and security (3)

Demilitarized zones



http://www.techrepublic.com/article/solutionbase-strengthen-network-defenses-by-using-a-dmz/

- Idea: Separate internal network as much as possible from the hostile internet.
- DMZ is not as hostile as the internet.
- DMZ and internal network are two separate networks. Compromising a machine in the DMZ is not the same as compromising the internal network.
- Place public servers in the DMZ – and place internal machines in the internal network: Compromising them requires two steps.

Example: Tradeoffs

- Problem description
  - Semi-automated surveillance and burglary detection
  - Houses with wireless sensors and wired cameras
  - Cameras can be monitored by customers
  - Not all alarms are actual burglary attempts
  - Notification of security personnel in cases of emergency
  - Customers and security officers use mobile clients
  - Security service company serves multiple customers

Requirements

Key functional requirements:

- System must **aggregate** sensor and camera **measurements**
- System must **detect emergency situations**
- System must **notify security personnel** in case of emergency
- System must be **accessible through mobile devices**
- System can be **monitored by customers**

Key non-functional requirements:

- The system must be **secure** and **reliable**
- The system must be **scalable**
- The system must be **privacy-preserving**
- The system communication must be **fast** and **cost-efficient**

A little software architecture example (1)

So… what do we make out of this?

- Need to make requirements more precise and ideally ***measurable***
- Need to map functional requirements to components
- Mapping must satisfy non-functional constraints
- Need to balance conflicting requirements and goals
- Need to be aware of conflicts and dependencies

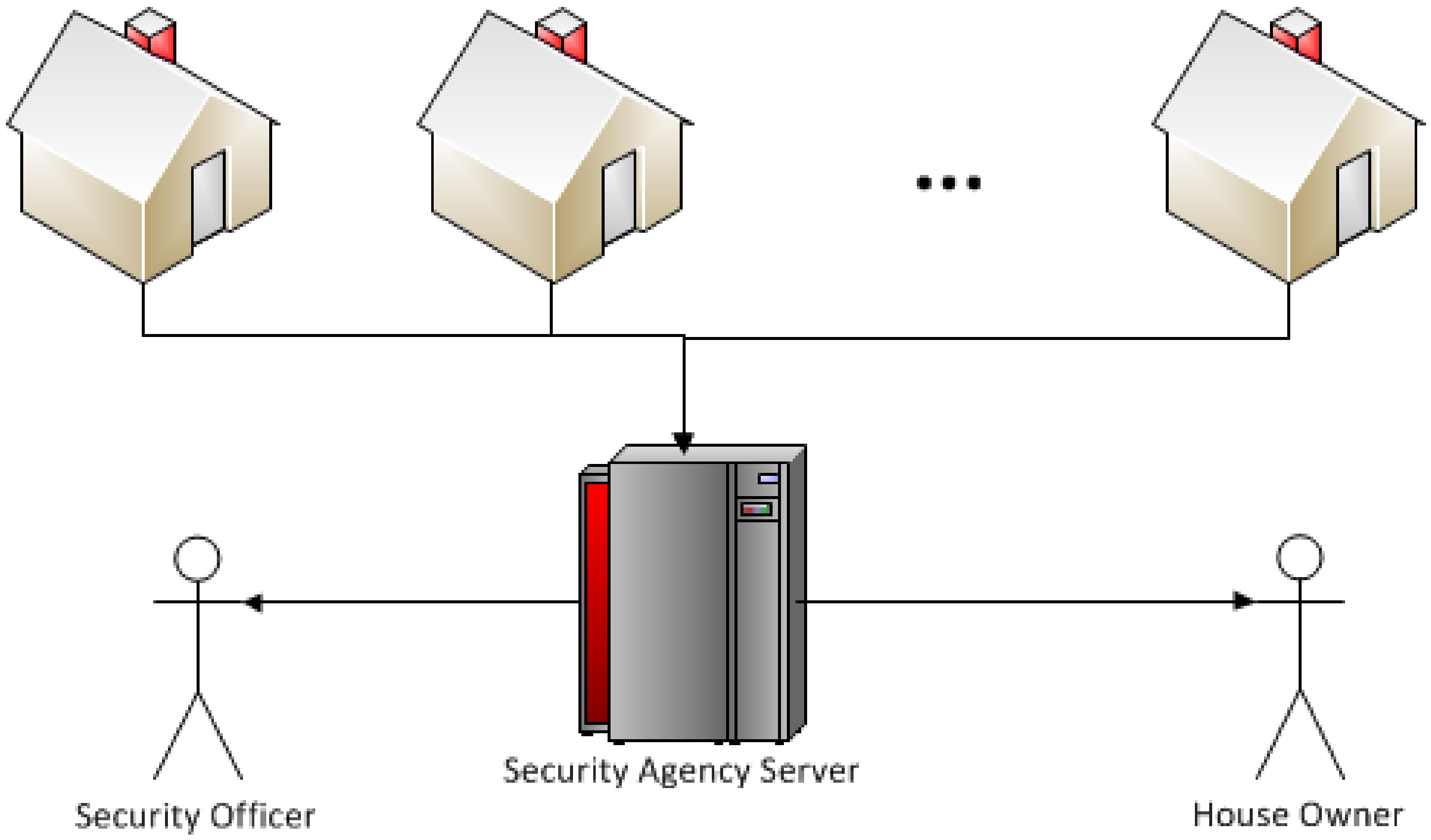
A little software architecture example (2)

So… what do we make out of this?

- Infinitely many architectural alternatives
- Alternatives vary in quality of requirement satisfaction
- Hard if not impossible to satisfy all requirements
- There is no perfect solution!
  → Need to prioritize and find compromises

- Let’s analyze some examples…

Context view

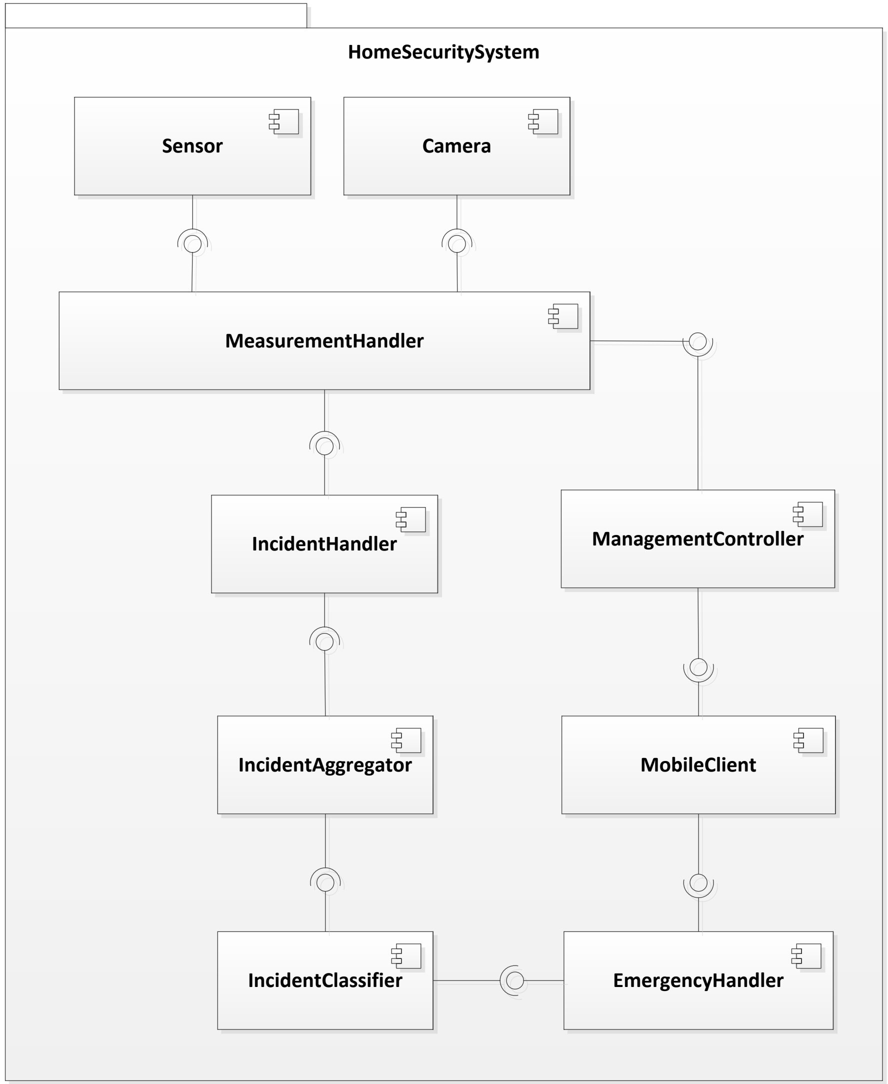


Component-and-connector view

Alternative A

- Decomposition is not deployment-oriented

- Rather tight coupling
  → scalability (-)
  → portability (-)
  → maintainability (-)
  → efficiency (+)

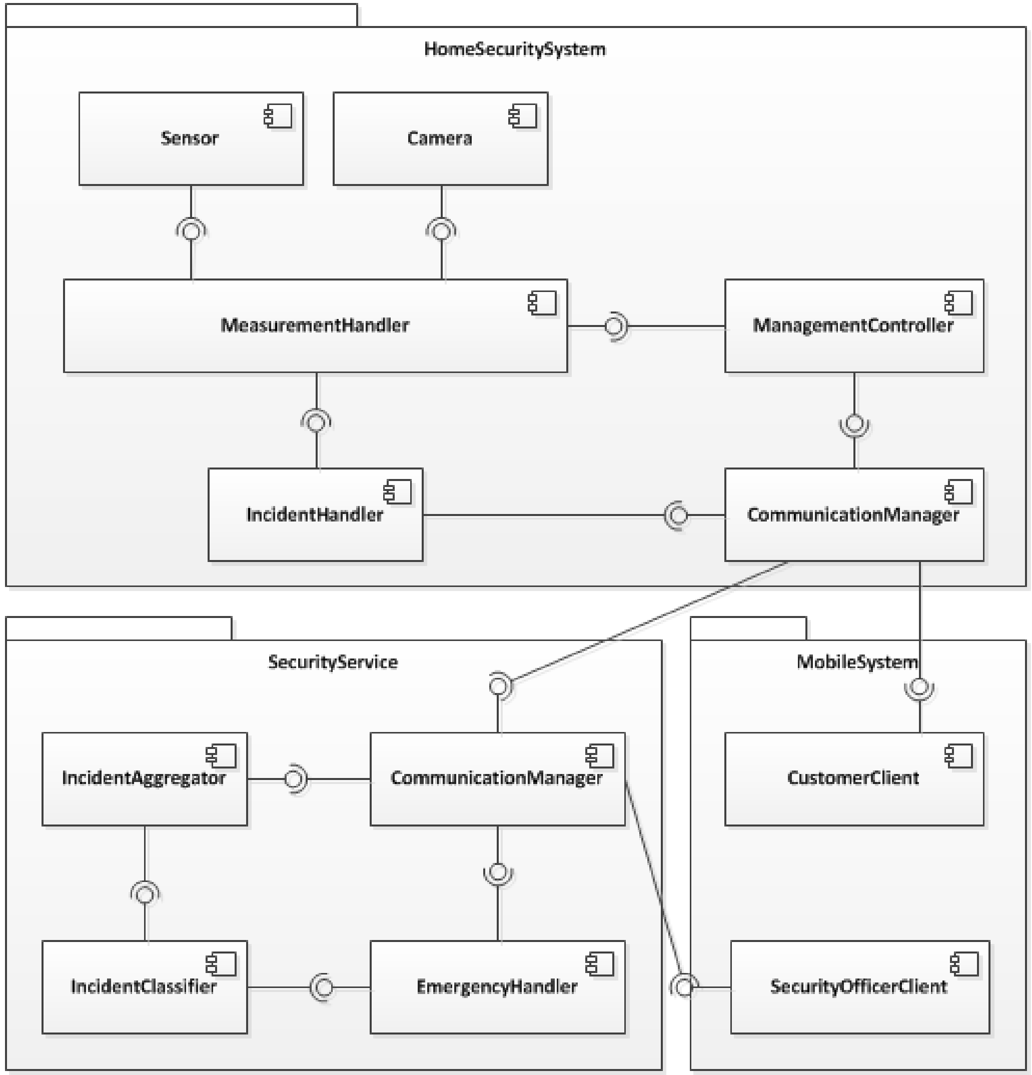


Component-and-connector view

Alternative B

Decomposition is deployment-oriented

→ scalability (+)
→ portability (+)
→ maintainability (+)
→ efficiency (-)

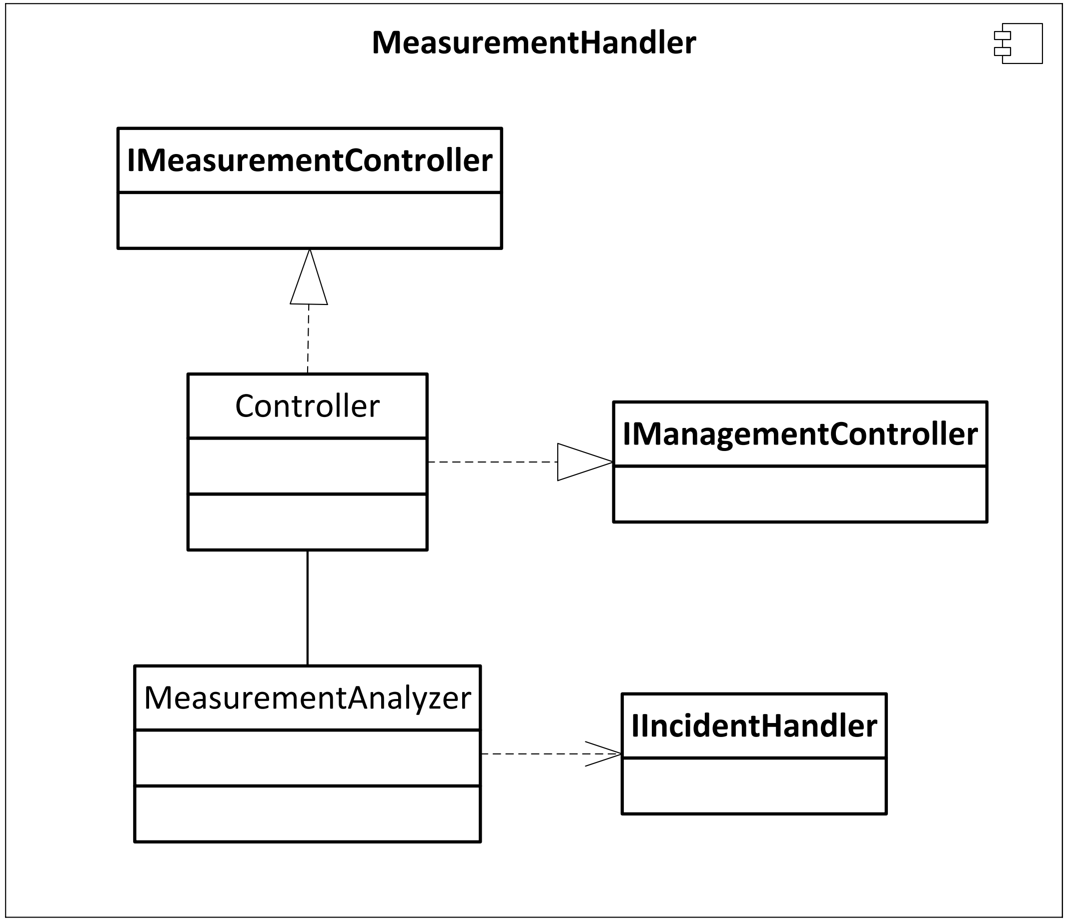


Module view

Alternative A

- Very simple structure
- Two “magic-classes”

- Barely decomposed
  → portability (-)
  → maintainability (-)
  → efficiency (+)

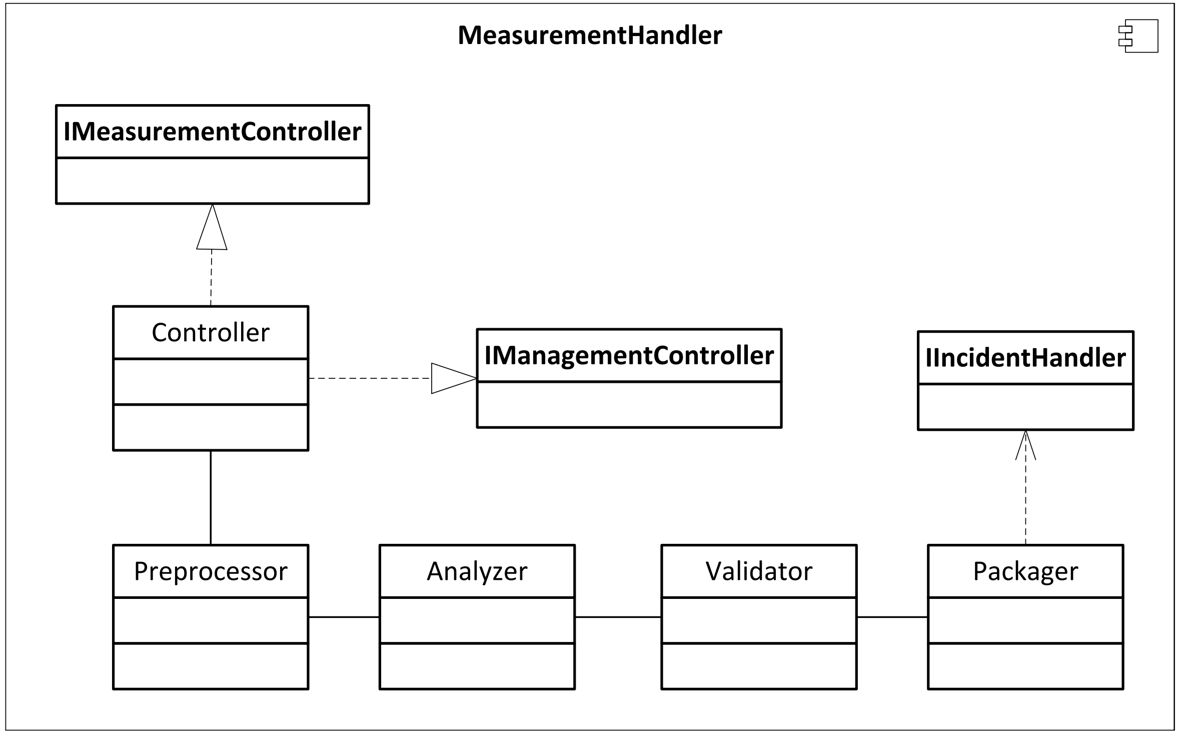


Module view

Alternative B

- Sequential processing
  → efficiency (+)

- Separation of concerns
  → scalability (+)
  → maintainability (+)

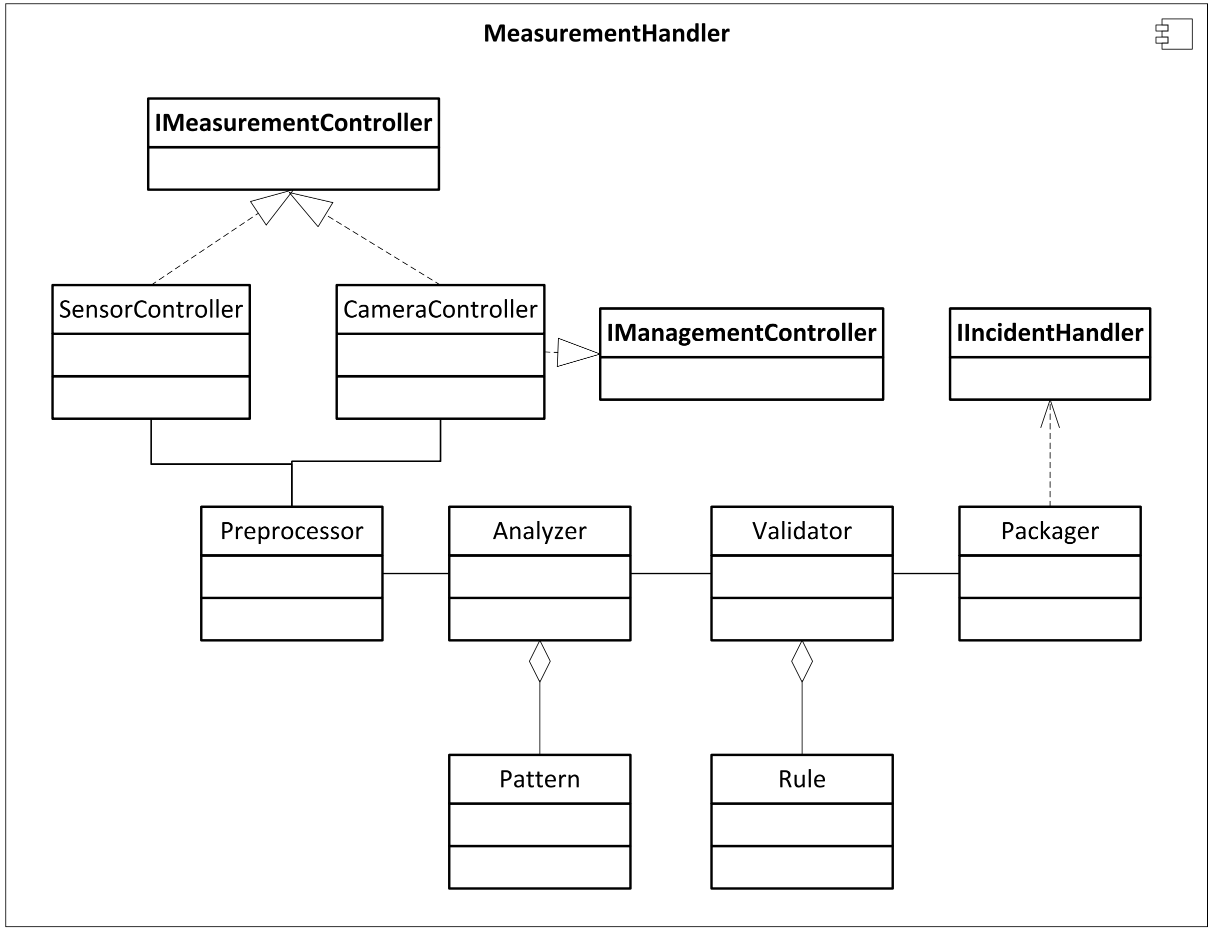


Module view

Alternative C

- Generic measurement controllers
  → maintainability (+)

- Pattern- and rule-aggregations
  → portability (+)
  → scalability (+)



Allocation view