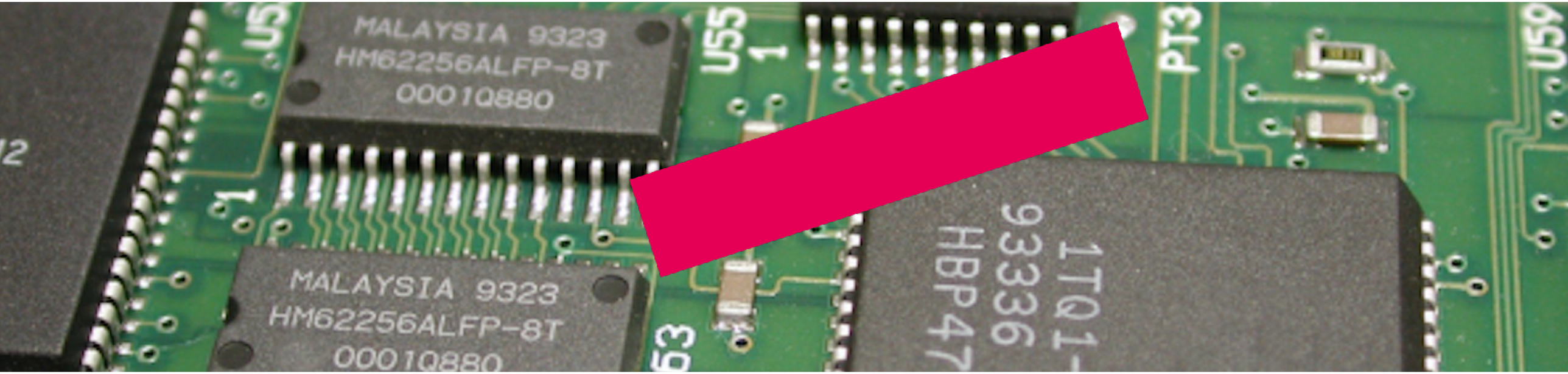
Embedded Systems Development - 6. Polymorphism



Electrical Engineering / Embedded Systems
School of Engineering and Automotive

Johan.Korten@han.nl

HAN_UNIVERSITY
OF APPLIED SCIENCES

Schedule (exact info see #OO and roster at insite.han.nl)

		C++	UML
Step 1	Single responsibility	Scope, namespaces, string	Use cases / Class diagram
Step 2	Open-Closed Principle	Constructors, iterators, lambdas	Inheritance / Generalization
Step 3	Liskov Substitution Principle	lists, inline functions, default params	Activities
Step 4	Interface Segregation Principle	interfaces and abstract classes	Depencencies
Step 5	Dependency Inversion Principle	threads, callback	Sequence diagrams
Step 6	**Coupling and cohesion**	**polymorphism**	**Composition, packages**
Step 7	Embedded SOLID	constructors, destructors, const	

‘Exam’ preparation (there is no exam anymore though)…

On top of the topics from the previous slide (some items might be on both slides):

C++
- constructor and destructor (RAII)
- access (visibility: private / public / protected)
- inheritance: base class, derived class, virtual functions, override
- class / struct
- abstract / interface, Abstract Base Class, abstract member function = 0
- getter / setter
- const-correct
- composite and aggregate
- range based for-loop (foreach) for container classes
- std:: name space, std::vector

UML and concepts
- class diagram, composite and aggregate, inheritance
- stereotype / object / actor
- abstract class / interface
- package diagram
- sequence diagram
- polymorphism, polymorphic arrays and vectors
- use cases
- state diagrams, events and states (prior knowledge)

Note: topics from previous C/C++ courses are considered as existing prior knowledge.

Note: the five *SOLID* principles will not be explicitly asked during the exam but can be helpful during the exam and are considered standard practices for good software engineering.

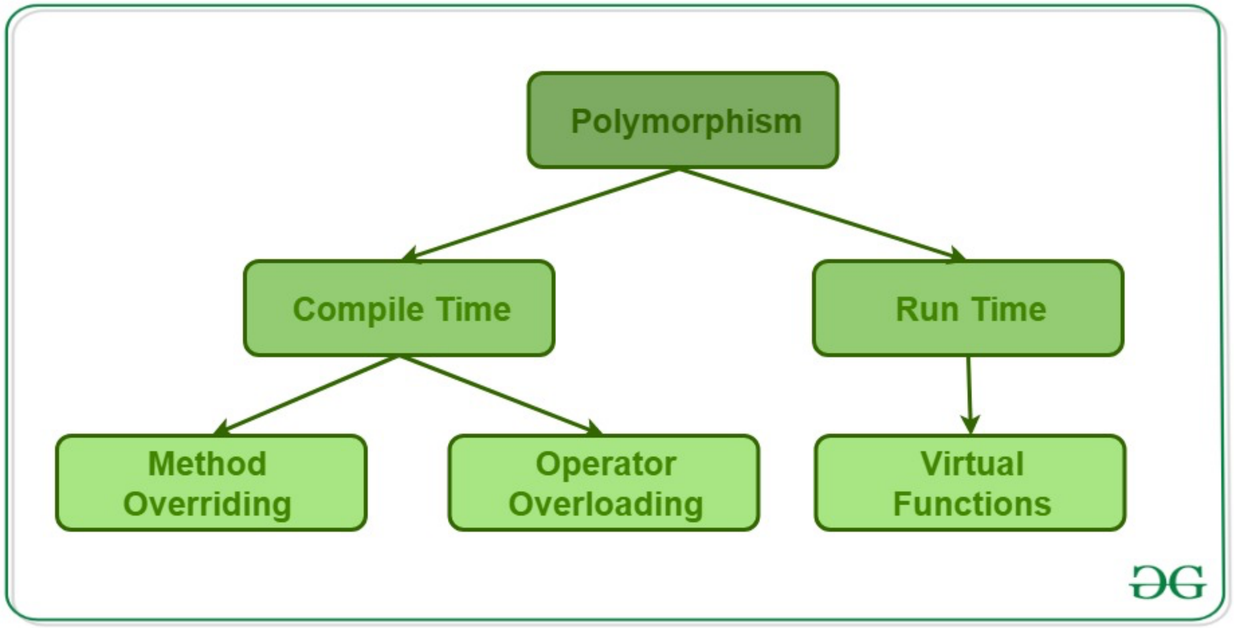
Constructs: Polymorphism

Polymorphism: the abstract *concept of dealing* with *multiple types* in a *uniform manner.*

Implementation: interfaces are a way to implement polymorphism. Code that interacts with an interface can interact with any type that provides that interface.

Remember: an interface is like a contract: it lays out the rules, but should not implement functionality (behavior) in contrast to an abstract class.

Constructs: Polymorphism



C++ Language: Polymorphic structures (vectors example)

You can e.g. create a vector of Pets / Dogs / whatever. (Addition to Dogs / Pets example from week 4).

```
vector <Dog*> someDogs;
someDogs.push_back(&goldie);
someDogs.push_back(&pluto);
someDogs.push_back(&stray);

cout << "\nLet all the dogs in someDogs bark: " << endl;

for (auto dog : someDogs) {
    dog->barks();
}
```

Let all the dogs in someDogs bark:

Digger barks to alert owner Boss!

Woof, woof, woof (meaning: I don't bite!)

Some stray dog: I always bark in fear of the dog catcher!

C++ Language: Polymorphic structures (arrays example)

You can also use polymorphism with arrays.

```
TwoDimensionShape *shapes[5];

shapes[0] = &Triangle("right", 8.0, 12.0);
shapes[1] = &Rectangle(10);
shapes[2] = &Rectangle(10, 4);
shapes[3] = &Triangle(7.0);
shapes[4] = &TwoDimensionShape(10, 20, "generic");

for(int i = 0; i < 5; i++) {
  cout << "object is " << shapes[i]->getName() << endl;

  cout << "Area is " << shapes[i]->area() << endl;

  cout << endl;
}
```

Note: again we use pointers to the object to store and reference the objects.

http://www.java2s.com/Code/Cpp/Class/Objectarraypolymorphism.htm

C++ Language: Polymorphism challenges

```
class School {
  private:
      std::vector<Person*> _people;

  public:
      void add(Person *person);
      void listPeople() const;
      void whatAreYouDoing() const;
      ~School();
};
```

How do we keep things tidy?!

C++ Language: Polymorphism challenges

```
class Person {
    public:
        virtual void whatsUp() = 0;
        const std::string getName() const;

        Person(const std::string &name) : _name(name) { ... }
        ~ Person();
    private:
        const std::string _name;
};
```

Ok, so whatsUp?!

Teachers and Students are fundamentally the same even though they might have some different behavior (now and then)…

C++ Language: Polymorphism challenges

```
class Student : public Person {

public:
    using Person::Person;
    void learns();
    void speaksUp();
    void whatsUp();
};
```

Ok, a student learns…

C++ Language: Polymorphism challenges

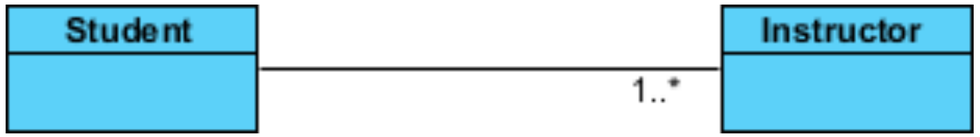
```
class Teacher : public Person {

public:
    using Person::Person;
    void teaches();
    void whatsUp();
    void speaksUp();
};
```

Ok, a teacher teaches…

UML relations: Association

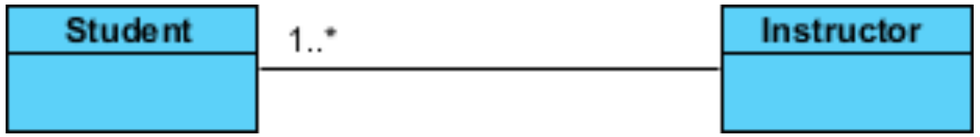
Let’s remember association



A student has one or more instructors.

UML relations: Association

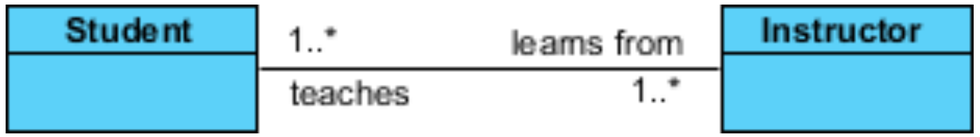
Let’s remember association



An instructor has one or more students.

UML relations: Association

Let’s remember association



Students have one or more instructors and instructors have one or more students.

UML relations: Association vs Aggregation vs Composition

Aggregation and Composition are specializations of Association.

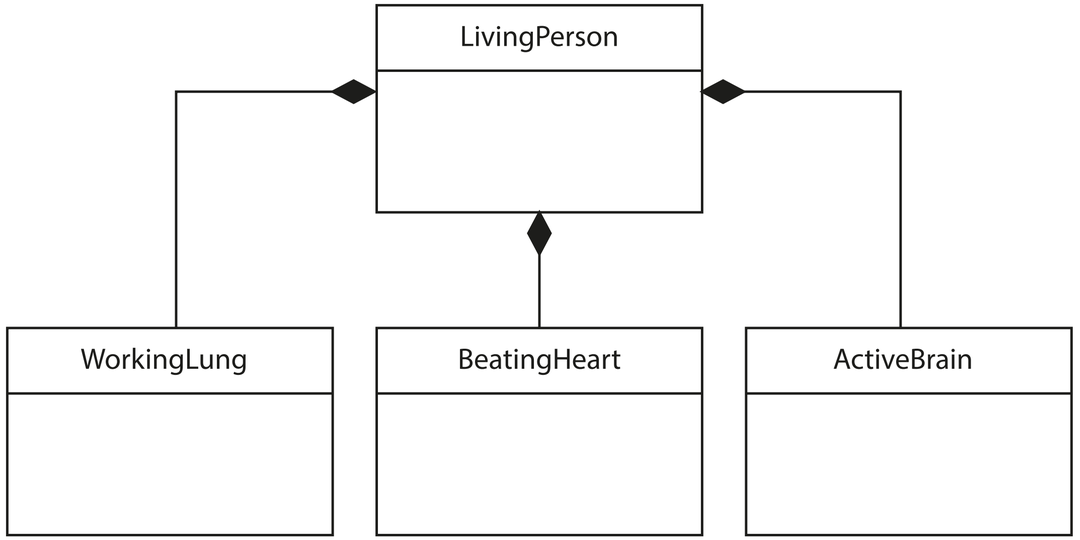
- Aggregation: child object can exist independently of parent object.

- Composition: child object can not exist independently of parent object.

UML relations: Association vs Aggregation vs Composition

Here we state that a *LivingPerson* is composed of WorkingLungs, BeatingHeart and ActiveBrains.

So this means (WorkingLungs + BeatingHeart + ActiveBrains) apparently is what makes (composes) a *LivingPerson.*

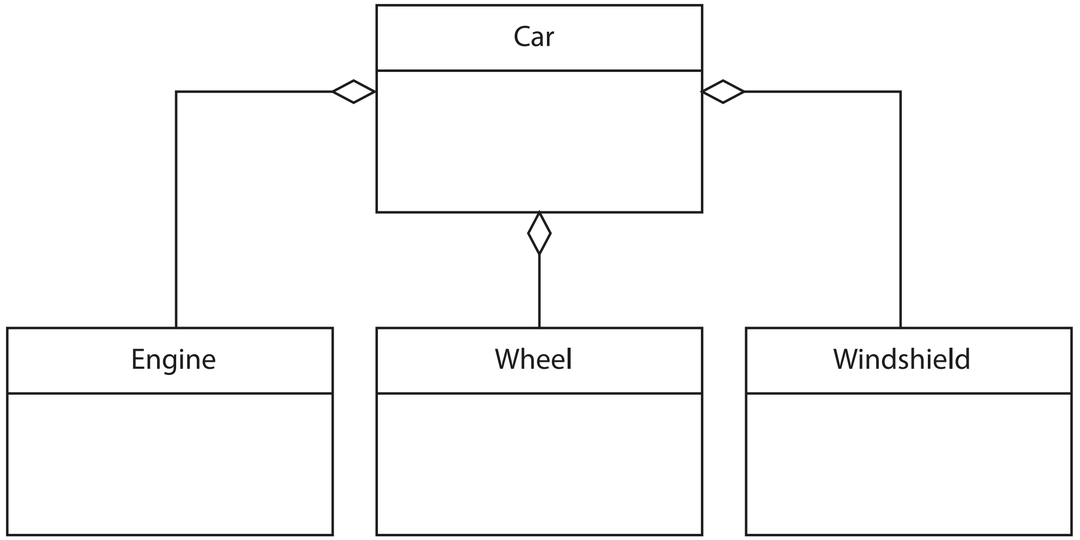


Obviously, without working lungs, beating heart and active brain a person is not a LivingPerson anymore.

UML relations: Association vs Aggregation vs Composition

Is an airfield without aircraft still an airfield?

Is a car without tires still a car?



UML relations: Association vs Aggregation vs Composition

Composition: child objects form the parent and the parent **cannot** exist without.

Aggregation: child objects are part of what makes the parent but the parent **can** exist without.

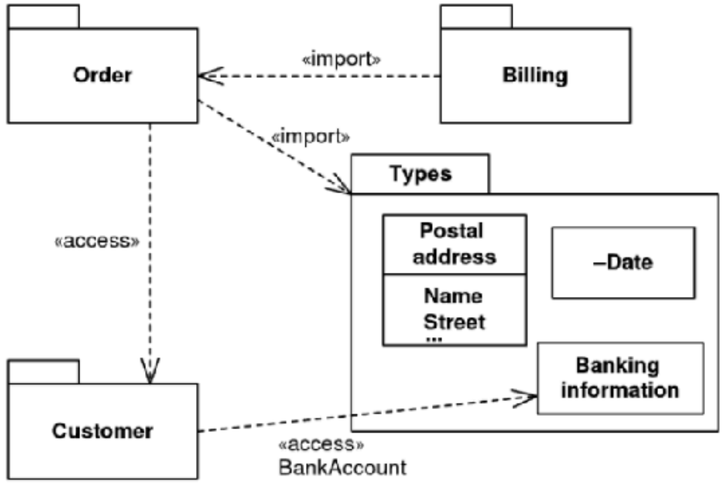
C++ Language: Aggregation and Composition

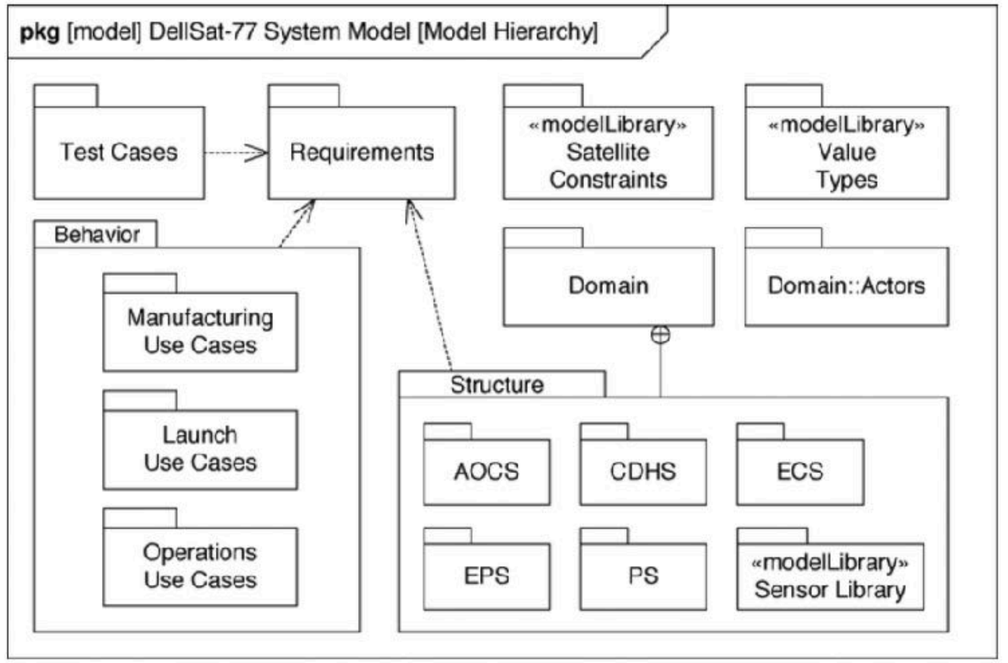
Composition: child objects form the parent and the parent cannot exist without.

Aggregation: child objects are part of what makes the parent but the parent can exist without.

UML Packages

The *package* groups model elements and builds a Namespace.





From UML 2 Certification Guide: Fundamental and Intermediate Exams by Weilkiens and Östereich p. 26

UML Packages

You can easily model the kind of dependencies between packages.

https://www.codeproject.com/Tips/1000719/High-Cohesion-Low-Coupling-using-SOLID-Principles

UML Packages

The *package* groups model elements and builds a Namespace.

UML Standard Stereotypes		
Stereotype	**UML Element**	**Description**
«call»	Dependency (usage)	Call dependency between operations or clases.
«create»	Dependency (usage)	The source element creates instances of the target element.
«instantiate»	Dependency (usage)	The source element creates instances of the target element. *Note*: This description is identical to the one of «create».
«responsibility»	Dependency (usage)	The source element is responsible for the target element.
«send»	Dependency (usage)	The source element is an operation and the target element is a signal sent by that operation.
«derive»	Abstraction	The source element can, for instance, be derived from the target element by a calculation
«refine»	Abstraction	A refinement relationship (e.g., between a design element and a pertaining analysis element).
«trace»	Abstraction	Serves to trace of requirements.
«script»	Artifact	A script file (can be executed on a computer).
«auxiliary»	Class	Classes that support other classes («focus»).
«focus»	Class	Classes contain the primary logic. See «auxiliary».
«implementationClass»	Class	An implementation class specially designed for a programming language, where an object may belong to one class only.
«metaclass»	Class	A class with instances that are, in turn, classes.
«type»	Class	Types define a set of operations and attributes, and they are generally abstract.
«utility»	Class	Utility classes are collections of global variables and functions, which are grouped into a class, where they are defined as class attributes/operations.
«buildComponent»	Component	An organizationally motivated component.
«implement»	Component	A component that contains only implementation, no specification.
«framework»	Package	A package that contains Framework elements.
«modelLibrary»	Package	A package that contains model elements, which are reused in other packages.
«create»	Behavioral feature	A property that creates instances of the class to which it belongs (e.g., constructor).
«destroy»	Behavioral feature	A property that destroys instances of the class to which it belongs (e.g., destructor).

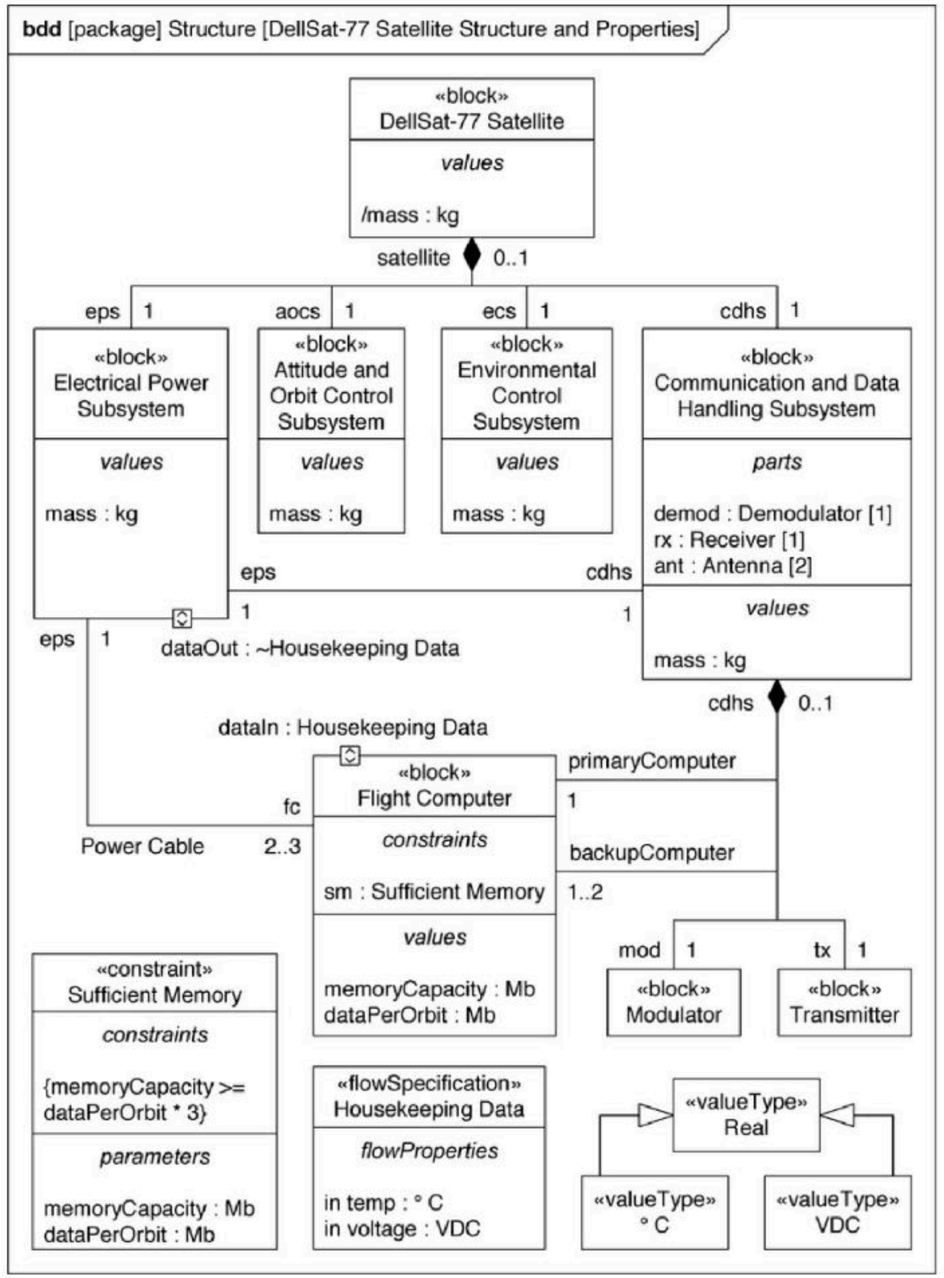
From UML 2 Certification Guide: Fundamental and Intermediate Exams by Weilkiens and Östereich p. 21

Note in SysML there are also Block Definition Diagrams (BDD's)

- When you want to focus on the *relations* between blocks or packages use a Package diagram
- When you want to focus on definitions use a BDD

From: SysML distilled by Delligatti, 2014.

BDD



From: SysML distilled by Delligatti, 2014.

Coupling and Cohesion

“If changing one module in a program requires changing another module, then coupling exists.”

Martin Fowler

https://martinfowler.com/ieeeSoftware/coupling.pdf

Coupling and Cohesion

Basic rules:

- Strive for Strong cohesion within modules (classes etc)

- Strive for Loose coupling between modules.

https://martinfowler.com/ieeeSoftware/coupling.pdf

Coupling and Cohesion in C++

It is not always easy to implement this principle even though it is an important one!

https://www.codeproject.com/Tips/1000719/High-Cohesion-Low-Coupling-using-SOLID-Principles

C++ Terminal application Case Study

Using (e.g.) Arduino
Adhering to SOLID-principles

HAN_UNIVERSITY
OF APPLIED SCIENCES

An Arduino Terminal application that adheres to the SOLID-principles.

A. Give a general idea:

- How can I implement a terminal application for Arduino?

B. Nudge the AI towards SOLID:

- What would be the best way to implement a terminal program that adheres to the SOLID principles?

C. Get basic code:

- Could you provide me with example code for that for C++ in Arduino?

D. Give additional directions:

- How do I implement the start method from userinterface as a callback method in the main program?

E. Improvements:

- How can I change it so it becomes non-blocking?

- How can I make this into a library?

- Could this terminal code be further perfected using a Singleton pattern or do you suggest other design patterns?

An Arduino Terminal application that adheres to the SOLID-principles.

A. Give a general idea:

- How can I implement a terminal application for Arduino?

B. Nudge the AI towards SOLID:

- What would be the best way to implement a terminal program that adheres to the SOLID principles?

C. Get basic code:

- Could you provide me with example code for that for C++ in Arduino?

D. Give additional directions:

- How do I implement the start method from userinterface as a callback method in the main program?

E. Improvements:

- How can I change it so it becomes non-blocking?

- How can I make this into a library?

- Could this terminal code be further perfected using a Singleton pattern or do you suggest other design patterns?

Terminal Application: SRP

1. Single Responsibility Principle (SRP)

Each class should have a single responsibility:

- **ConnectionManager**: Manages the opening, closing, and state of the serial connection.

- **IOHandler**: Deals with input/output operations; reading from and writing to the serial connection.

- **UserInterface**: Handles user interactions, display feedback, and error messages.

Terminal Application: OCP

2. Open/Closed Principle (OCP)

Software entities should be open for extension but closed for modification.

Use *interfaces* or *abstract classes* for components, so if you decide to change the behavior or add new features, you can create new implementations without modifying existing code.

For example, if you want to add a new communication protocol (like I2C or SPI), you can implement a new connection manager without touching the existing SerialConnectionManager.

Terminal Application: LSP

3. Liskov Substitution Principle (LSP)

Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

Derived classes must be substitutable for their base classes. If you have an abstract ***ConnectionManager*** class, then derived classes like ***SerialConnectionManager*** or ***I2CConnectionManager*** should be able to replace the base class without any hitches.

Terminal Application: ISP

4. Interface Segregation Principle (ISP)

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

Break down the interfaces to make them specific to the client's needs. For instance, if you have methods related to connection settings, don't force IO-related classes to implement or know about these methods.

Terminal Application: DIP

5. Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Rather than the ***UserInterface*** class directly creating and using a ***SerialConnectionManager***, it should depend on an abstraction like an ***IConnectionManager*** interface. The specific implementation (e.g., ***SerialConnectionManager***) is injected into ***UserInterface*** via constructor or method, enabling the flexibility to change the communication method without altering the higher-level module.

Terminal Application: An Example Structure

1. **IConnectionManager** (interface)
   - *connect()*
   - *disconnect()*
   - *isConnected()*
2. **SerialConnectionManager** (implements ***IConnectionManager***)
3. ***IOHandler***
   - Constructor accepts an ***IConnectionManager***
   - *read()*
   - *write(data)*
4. **UserInterface**
   - Constructor accepts an ***IOHandler***
   - start()
   - displayMessage(message)
   - getUserInput()

Terminal Application: An Example Structure

To create a terminal program:

- Instantiate the specific connection manager (SerialConnectionManager).
- Pass the connection manager to the IOHandler.
- Pass the IOHandler to the UserInterface.
- Call start() on the UserInterface.

An Arduino Terminal application that adheres to the SOLID-principles.

A. Give a general idea:

- How can I implement a terminal application for Arduino?

B. Nudge the AI towards SOLID:

- What would be the best way to implement a terminal program that adheres to the SOLID principles?

C. Get basic code:

- Could you provide me with example code for that for C++ in Arduino?

D. Give additional directions:

- How do I implement the start method from userinterface as a callback method in the main program?

E. Improvements:

- How can I change it so it becomes non-blocking?

- How can I make this into a library?

- Could this terminal code be further perfected using a Singleton pattern or do you suggest other design patterns?

An Arduino Terminal application that adheres to the SOLID-principles.

A. Give a general idea:

- How can I implement a terminal application for Arduino?

B. Nudge the AI towards SOLID:

- What would be the best way to implement a terminal program that adheres to the SOLID principles?

C. Get basic code:

- Could you provide me with example code for that for C++ in Arduino?

D. Give additional directions:

- How do I implement the start method from userinterface as a callback method in the main program?

E. Improvements:

- How can I change it so it becomes non-blocking?

- How can I make this into a library?

- Could this terminal code be further perfected using a Singleton pattern or do you suggest other design patterns?

An Arduino Terminal application that adheres to the SOLID-principles.

A. Give a general idea:

- How can I implement a terminal application for Arduino?

B. Nudge the AI towards SOLID:

- What would be the best way to implement a terminal program that adheres to the SOLID principles?

C. Get basic code:

- Could you provide me with example code for that for C++ in Arduino?

D. Give additional directions:

- How do I implement the start method from userinterface as a callback method in the main program?

E. Improvements:

- How can I change it so it becomes non-blocking? (call-back)

- How can I make this into a library?

- Could this terminal code be further perfected using a Singleton pattern or do you suggest other design patterns?

An Arduino Terminal application that adheres to the SOLID-principles.

A. Give a general idea:

- How can I implement a terminal application for Arduino?

B. Nudge the AI towards SOLID:

- What would be the best way to implement a terminal program that adheres to the SOLID principles?

C. Get basic code:

- Could you provide me with example code for that for C++ in Arduino?

D. Give additional directions:

- How do I implement the start method from userinterface as a callback method in the main program?

E. Improvements:

- How can I change it so it becomes non-blocking? (call-back)

- How can I make this into a library?

- Could this terminal code be further perfected using a Singleton pattern or do you suggest other design patterns?

An Arduino Terminal application that adheres to the SOLID-principles.

A. Give a general idea:

- How can I implement a terminal application for Arduino?

B. Nudge the AI towards SOLID:

- What would be the best way to implement a terminal program that adheres to the SOLID principles?

C. Get basic code:

- Could you provide me with example code for that for C++ in Arduino?

D. Give additional directions:

- How do I implement the start method from userinterface as a callback method in the main program?

E. Improvements:

- How can I change it so it becomes non-blocking? (call-back)

- How can I make this into a library?

- Could this terminal code be further perfected using a Singleton pattern or do you suggest other design patterns?

An Arduino Terminal application that adheres to the SOLID-principles.

Coupling Between Components

Components like IOHandler and UserInterface are dependent on IConnectionManager.

They don't directly know about SerialConnectionManager.

This is a good design because these classes are decoupled from the specific implementation of the connection manager.

Source: OpenAI ChatGPT 4

An Arduino Terminal application that adheres to the SOLID-principles.

Benefits for Loose Coupling

The interface-based design (IConnectionManager).

If in the future you decided to have a *BluetoothConnectionManager* or *WiFiConnectionManager*, you could do so without changing the *IOHandler* or *UserInterface* classes, as long as these new managers also implement the *IConnectionManager* interface. This is the essence of loose coupling.

Source: OpenAI ChatGPT 4

An Arduino Terminal application that adheres to the SOLID-principles.

Enhancing Loose Coupling

The interface-based design (IConnectionManager).

If you're aiming to further enhance loose coupling in the system, consider patterns like the Factory pattern (to dynamically create objects without specifying the exact class of object that will be created) or the Strategy pattern (to define a family of algorithms and make them interchangeable).

These patterns can help reduce dependencies between classes and make the system more modular and extensible.

Source: OpenAI ChatGPT 4

An Arduino Terminal application that adheres to the SOLID-principles.

Overhead: Memory

Dynamic Memory (Heap) Overhead: Introducing the Singleton pattern with dynamic allocation (new) adds memory overhead. The dynamically allocated object remains in memory until the program terminates or until explicitly deleted.

Static Memory (Stack) Overhead: Each interface and class definition, if not optimized out by the compiler, may add some overhead. But this is usually minimal and is more about the code footprint (size) than active memory consumption.

Source: OpenAI ChatGPT 4

An Arduino Terminal application that adheres to the SOLID-principles.

Overhead: Code Size

Modular design and use of design patterns can lead to a larger number of lines of code and, consequently, a larger binary size.

Depending on the microcontroller's storage capacity, this might be a consideration.

Source: OpenAI ChatGPT 4

An Arduino Terminal application that adheres to the SOLID-principles.

Overhead: Complexity

Introducing design patterns and interfaces can make the codebase more complex, especially for developers unfamiliar with these concepts.

This can impact the maintainability of the code, though one could argue that a well-structured and modular codebase improves maintainability in the long run.

Source: OpenAI ChatGPT 4

An Arduino Terminal application that adheres to the SOLID-principles.

Overhead: Flexibility and Extensibility Benefits

The overhead introduced often provides benefits in terms of flexibility and extensibility.

For example, using interfaces allows you to easily swap out components or add new features without impacting existing code.

This modularity can save time and reduce bugs in future developments.

Source: OpenAI ChatGPT 4

An Arduino Terminal application that adheres to the SOLID-principles.

Overhead: Other

Initialization times might be slightly affected, especially if the Singleton pattern or other patterns require more complex initialization logic.

There might be additional overhead when working with tools that aren't optimized for object-oriented designs or which have limited support for C++ features.

Source: OpenAI ChatGPT 4

An Arduino Terminal application that adheres to the SOLID-principles.

Conclusion

While there is some overhead associated with the design decisions we've discussed, it's essential to balance these overheads with the benefits. For smaller, simpler projects, the overhead might not be justified, and a more straightforward procedural approach might suffice. However, for larger projects or those expected to grow and evolve, the flexibility, modularity, and maintainability offered by these design principles can far outweigh the overheads, leading to more manageable and scalable software in the long run.

Source: OpenAI ChatGPT 4

Links on UML

SysML Distilled: A Brief Guide to the Systems Modeling Language, Delligatti, 2014.

Systems Engineering mit SysML/UML: Anforderungen, Analyse, Architektur, Weilkiens 2014.

C++ Good programming practices

https://martinfowler.com/ieeeSoftware/coupling.pdf

More on GitHub:

https://youtu.be/3VEU88T1bVk

Any questions?