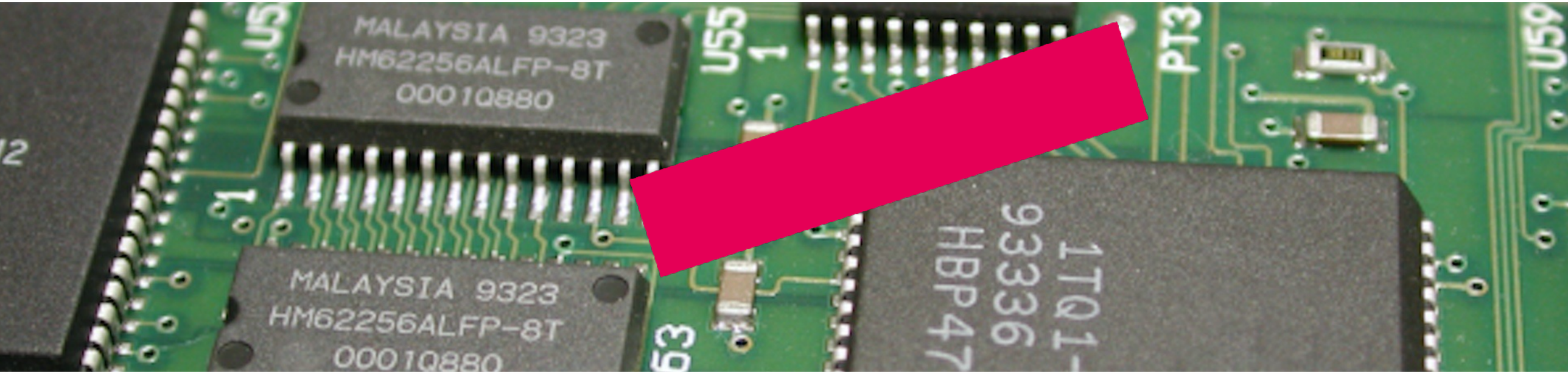
Embedded Systems Development - 6. Polymorphism



Electrical Engineering / Embedded Systems
Faculty of Engineering

Ruud.Elsinghorst@han.nl
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	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		what is left… exam preparation	Use cases

Note: subject to changes as we go…

Exam preparation

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.

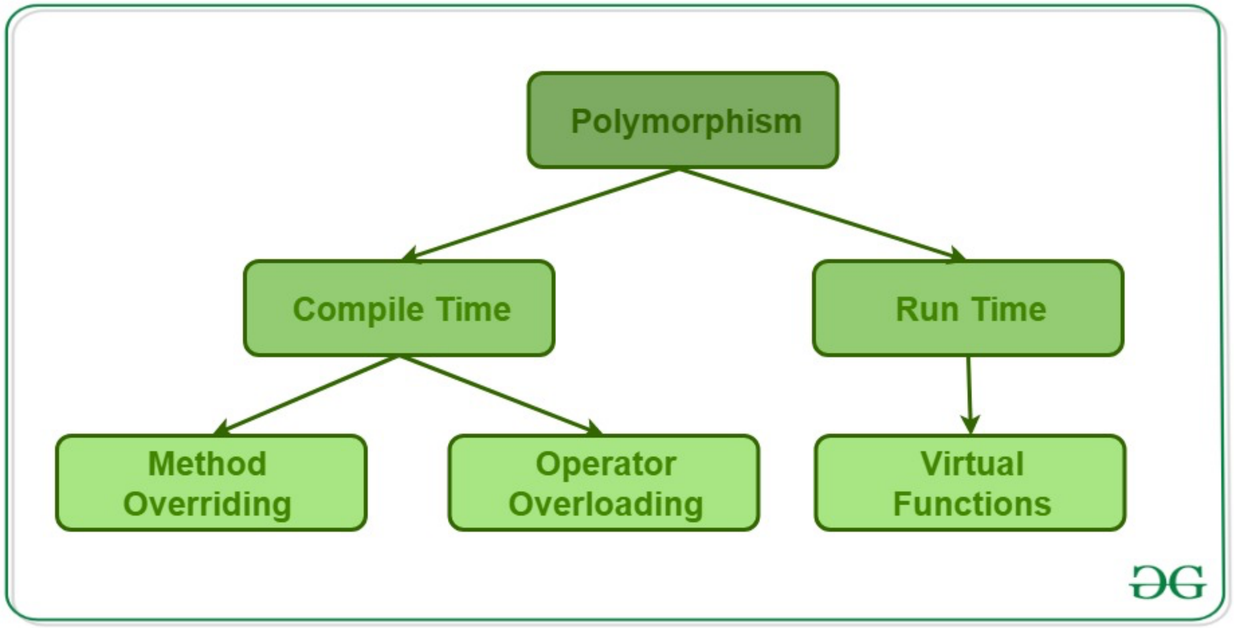
Constructs: Polymorphism

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

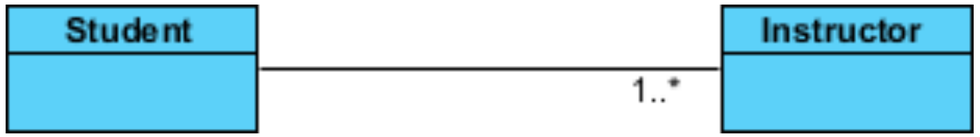
```
class Teacher : public Person {

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

Ok, a teacher teaches…

Let’s switch to the example code.

UML relations: Association

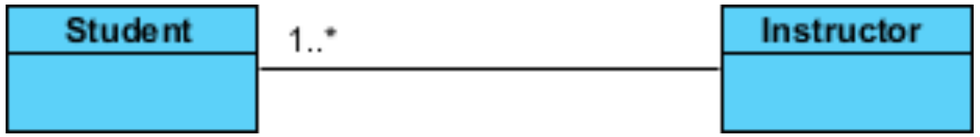
Let’s remember association



A student has one or more instructors.

UML relations: Association

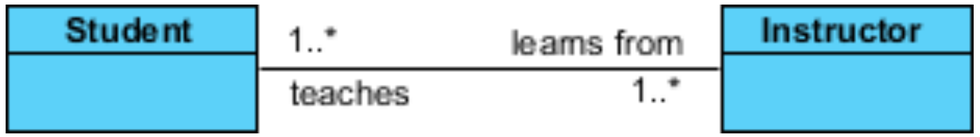
Let’s remember association



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

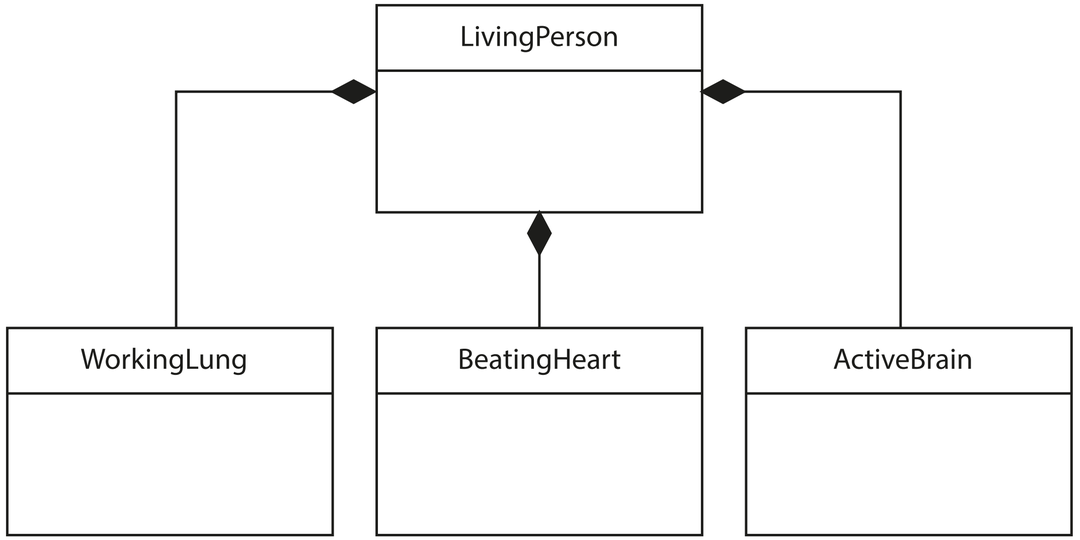
Aggregation and Composition are specializations of Association.

- 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.



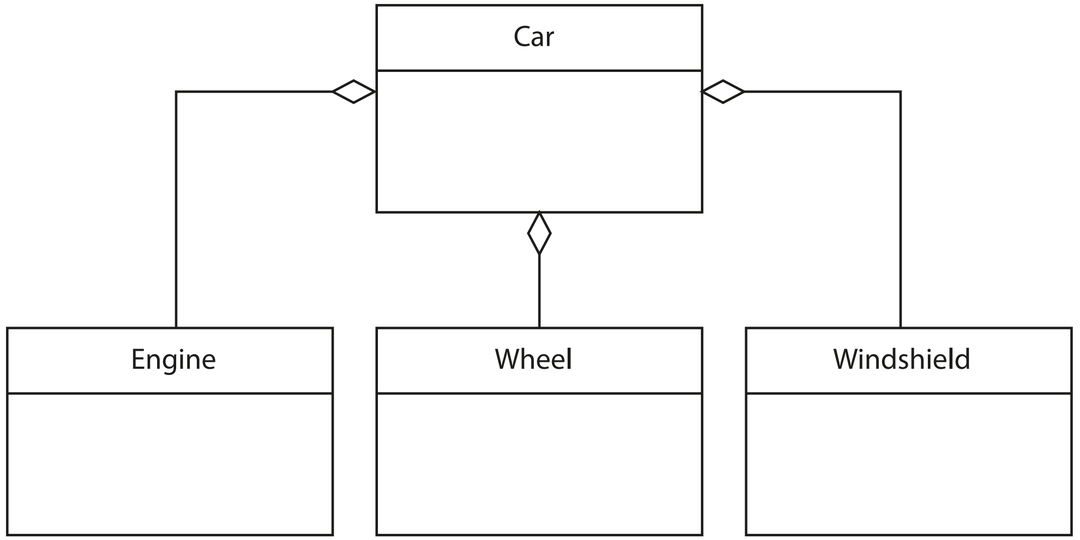
So this means (WorkingLungs + BeatingHeart + ActiveBrains) apparently is what makes (composes) a *LivingPerson.*

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.

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

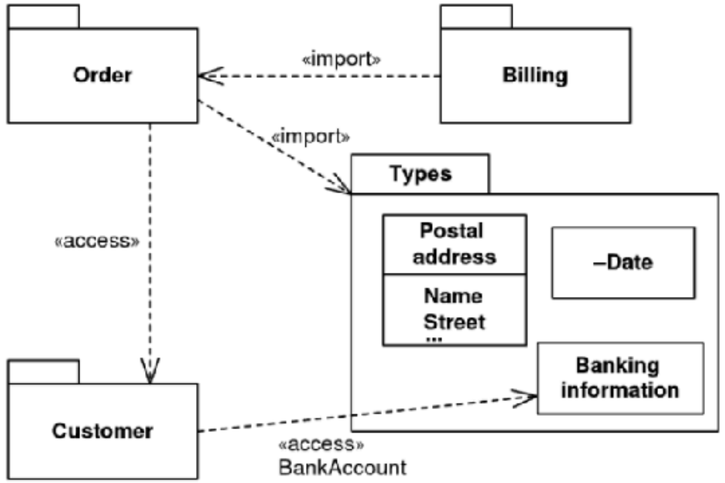
Coupling and Cohesion in C++

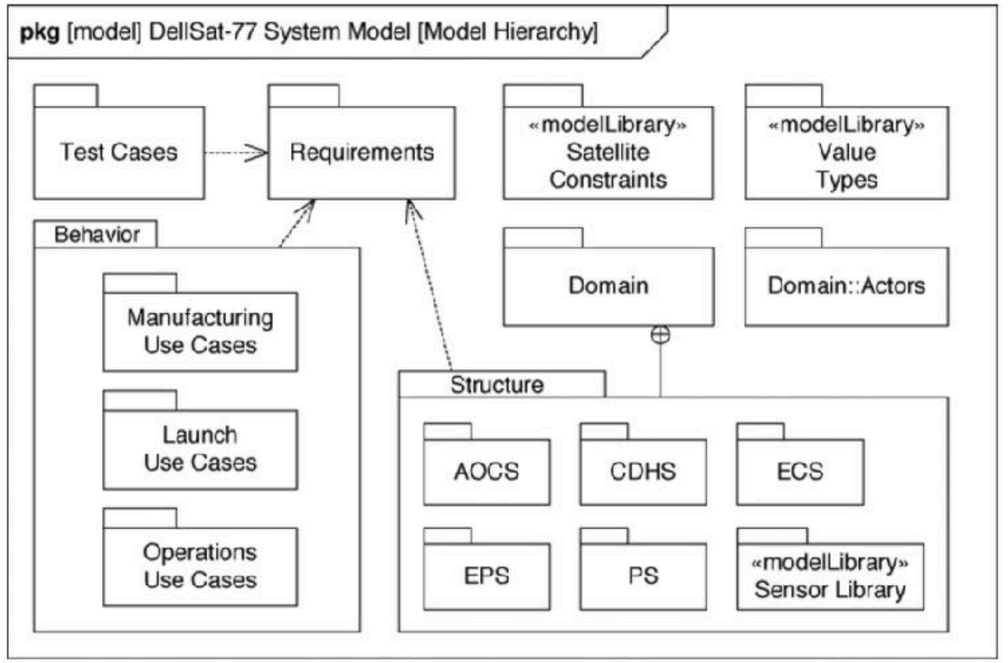
It is not always easy to implement this principle even though it is an important one!

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

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

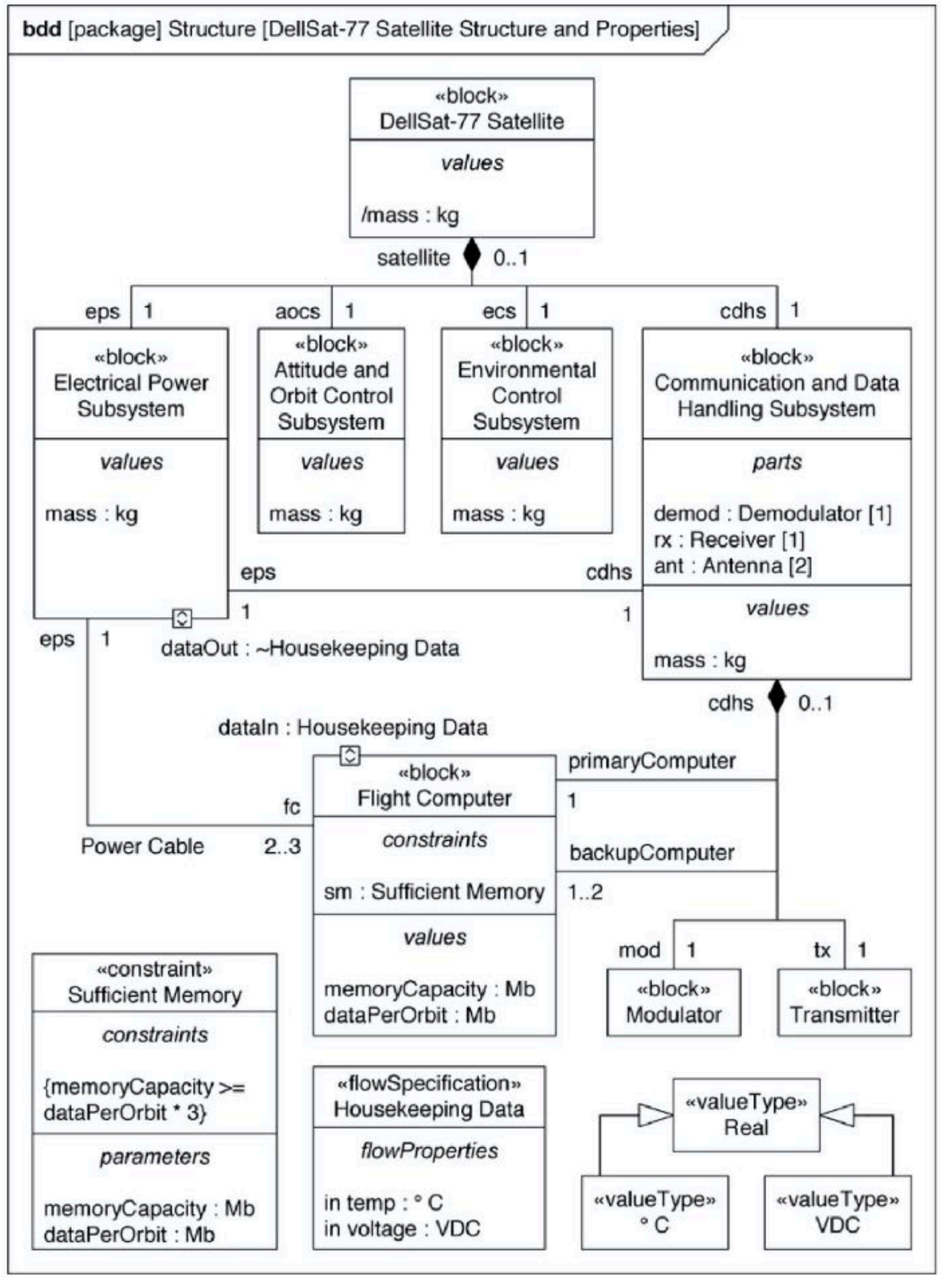
Note in SysML there are also Block Definition Diagrams (BDD's)

- 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 (Not for the exam)



From: SysML distilled by Delligatti, 2014.

Just to be sure: Git configuration

Git

Initial Git setup:

```
git config --global user.name "YOUR_USERNAME"

git config --global user.email "your_email_address@example.com"

git config --global --list

git config --global pull.ff only
```

https://docs.gitlab.com/ee/gitlab-basics/start-using-git.html

Just to be sure: Git clone - making a repository

- Create online Git repository
- cd to the local folder you want to use for that online repo
- copy the https address of your online repo to the clipboard

Issue:

```
git clone https://gitlab.com/gitlab-org/gitlab.git
```

You are set to use that folder from now on as a Git repository

Just to be sure: Git commit and push to remote repository

The 1-2-3 steps to push your local changes…

1. Add all files in current folder to the local repo:

```
git add *
```

2. Commit the local files to the repo for pushing later on

```
git commit –m "Reason for committing"
```

// Note changes are then still only in your local repository…

3. Add the changes to the remote repository:

```
git push
```

Just to be sure: Git fetch / pull

Let’s say you are working together using the same repo:

1. Start your day by issuing

```
git fetch
```

```
git merge remotename/branchname
```

or issue:

```
git pull
```

See more on potential issues with git pull at

https://stackoverflow.com/questions/15316601/in-what-cases-could-git-pull-be-harmful

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?