**Domain model is an object model of the domain that incorporates both behavior and data**.  
The domain model is frequently opposed to a model where you have specific classes to bear data and some other specific classes to bear behavior/processing.

If I take your Income class, it looks like more as a class that holds properties/data than an domain model with a real behavior.

public boolean isPositive(){

return amount > 0 ? true : false;

}

is a kind of utility function that has no relation with the model.  
You could put that in a Math class.

I will try to give you a domain model example and then the version where the model separates data and processing.

Suppose in the requirements of the domain of the application you are modeling, we need to add a bonus for incomes. This bonus may take place in winter for Christmas for example (but why not for other events)

Rather than having a service class to do this processing, we let domain model objects perform the task.

Incomes, a high level object could iterate on Income instances and apply the bonus and we could have a bonus rule class that defines the bonus according to some input values.  
I introduce multiple classes since the idea is to allow each objects to collaborate according to their responsibilities.

Incomes :

public class Incomes {

List<Income> incomes = ...

....

public void applyBonus(BonusRule bonusRule){

for (Income income : incomes){

income.applyBonus(bonusRule);

}

}

Income :

public class Income {

private float amount;

...

public void applyBonus(BonusRule bonusRule){

float bonus = bonusRule.compute(this);

amount += bonus;

}

...

}

ChristmasRule :

public class ChristmasBonusRule implements BonusRule {

...

@Override

public float compute(Income income){

float bonus = ...

return bonus;

}

...

}

And finally, we could apply the processing in this way :

void foo(){

// create a domain object that has both behavior and data

Incomes incomes = ...;

// invoke a functional method on the object by passing another domain object

incomes.applyBonus(new ChristmasBonusRule());

}

In a design where you separate data and logic in distinct classes, it could look like more like that :

public class IncomeBonusService {

// stateless : no incomes data inside it

....

public void applyChristmasBonus(List<Income> incomes){

for (Income income : incomes){

// Christmas bonus computation here

float bonus = ...

income.setAmount(bonus + income.getAmount());

}

}

}

And we could apply the processing in this way :

// inject the service

@Autowired

IncomeBonusService incomeBonusService;

void foo(){

// create a domain object that has only data

List<Income> incomes = ...;

// invoke a service method by passing data as parameter

incomeBonusService.applyChristmasBonus(incomes);

}

A model design where the objects have no behavior (only getter/setter) is called [Anemic Domain Model](https://www.martinfowler.com/bliki/AnemicDomainModel.html).

Big differences between the two ways illustrated by this example :

Domain model :

* The objects are meaningful.
* Behavioral responsibility finely defined between classes.  
  So good isolation, testability and maintainability.  
  For example, adding/removing/unit-testing a BonusRule is easy.
* Objects responsible of their state.  
  Indeed, no need to provide setters as the object can itself update its state after collaborating with other objects.  
  We could see that in Amount.applyBonus() :

float bonus = bonusRule.compute(this); amount += bonus;

Anemic Domain Model :

* All the logic is in the service class.  
  So a single place to get the code.  
  With few lines, it is fine.  
  But note that this advantage has a certain limit because as the logic becomes big or complex, the best thing is often splitting the logic in multiple service classes.
* But whatever the number of Service classes you need, the whole logic is located in the service classes and not somewhere else. Which may ease development norms if we compare it to the domain model where the logic may be exploded in some different "types" of classes.
* Necessity to provide getter/setter for domain classes.  
  The domain is not responsible of its state and its invariant rules either.  
  So any class that depends on the domain class can "break" its state.

As a side note, some frameworks (for persistence, mapping, serialization, ...) rely by default on getter/setter.  
That's why this model, despite its drawbacks, leads in some projects.

Fowler's book that you cite refers to [Larman's book](https://rads.stackoverflow.com/amzn/click/com/0131489062) for introductory understanding and examples.

Interestingly, Larman's approach to domain modeling doesn't ever add behavior to domain classes.

There is a notion in a domain model that a class is *conceptual* and is not a software class, but that software classes are based on domain (conceptual) classes. Larman's approach to implementing behavior follows [responsibility driven design](https://en.m.wikipedia.org/wiki/Responsibility-driven_design), and GoF design patterns.

The domain model remains a separate element in the software development process. The good aspect of this way of modeling is you separate the problem from the solution. A domain model is supposed to be true to the problem (capture requirements from the problem domain without addressing implementation details).

Larman presents "operation contracts" as a way to assure behaviors are consistent inside a domain model. Again, contracts are supposed to be independent of a solution (an implementation). Contracts have postconditions that describe a constraint in the domain model after an operation has taken place. An example of a postcondition would be that when a customer completes a purchase in a store, the sale object is associated with each of the items the customer purchased.

The implementation of the business logic should respect the contracts (postconditions) defined for the domain model. Larman's approach with the [Controller GRASP pattern](https://en.wikipedia.org/wiki/GRASP_(object-oriented_design)#Controller) as well as other GRASP patterns ends up putting this logic in various classes (usually the domain layer, which is software classes inspired by conceptual classes in the domain model) or Façade (Controller) classes that handle the system operations.

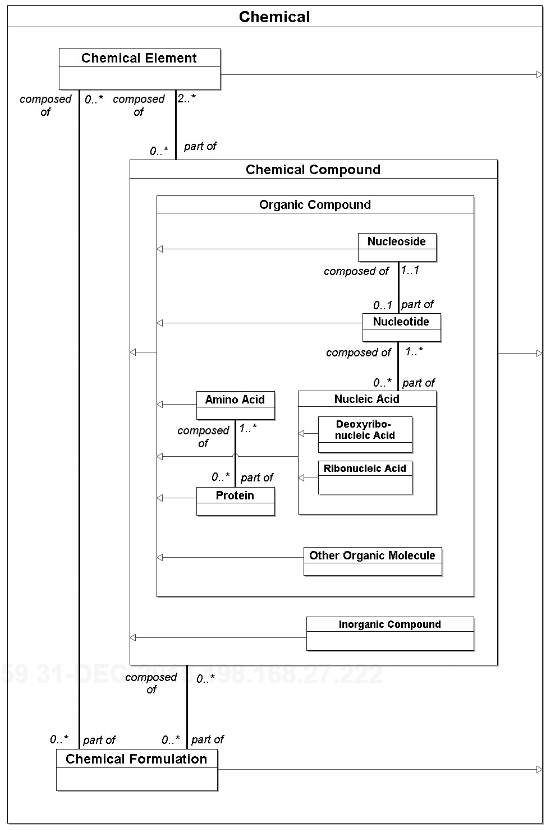
Larman's approach is more complicated than this explanation, but the point is that behavior is never only defined alone in the domain model as methods. Larman says many times that domain (conceptual) classes do not have methods, as they are *not* software classes.

Fowler's book also refers to another book that he wrote on [Analysis Patterns](http://martinfowler.com/books/ap.html) for examples.

The patterns come from various domains, including health care, financial trading, and accounting. Each of the patterns is described both textually and in a simple pre-UML notation (this book was written before the UML had stabilized into a usable form).

None of the examples in that book show software classes, that is with methods defined in a programming language (that I could find).

I know of at least [one book](https://rads.stackoverflow.com/amzn/click/com/1935504053) where domain models in fields such as molecular biology have been published in UML. Here's an example (note the UML is modified -- sub-type boxes are shown in super-type boxes -- to save space):

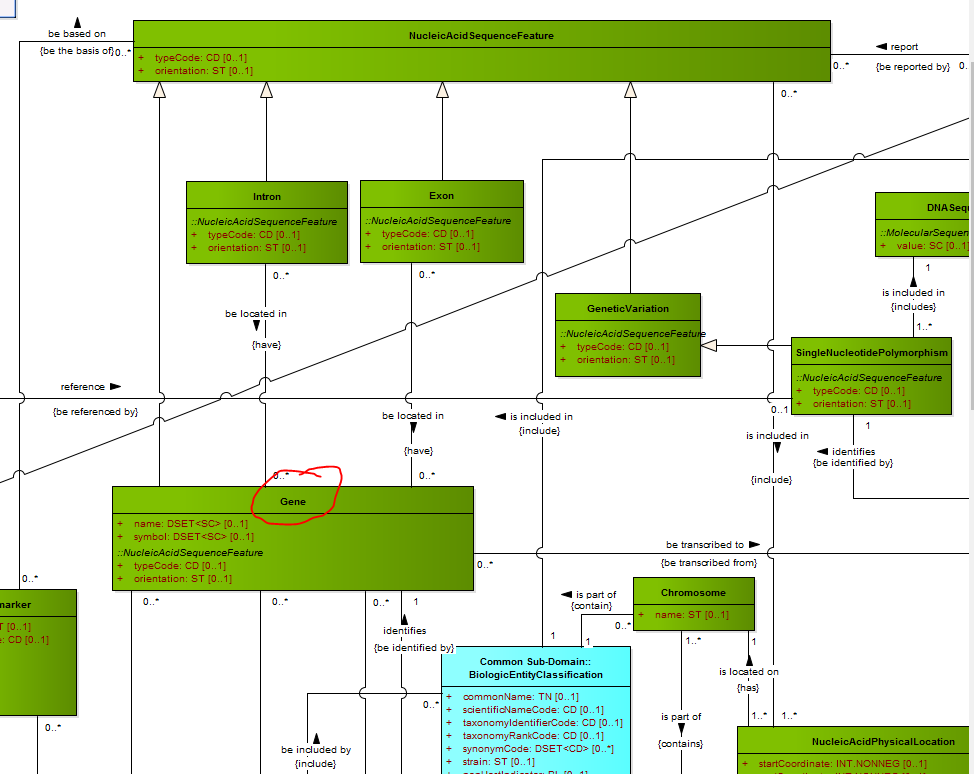
[](https://i.stack.imgur.com/s3PpZ.png)

The book above does not model behaviors, probably because they really depend on the requirements of the software application. These models capture some business rules, such as:

* Each **Chemical Formulation** must be composed of 2 or more of either **Chemical Elements** or **Chemical Compounds**, or both.

But the models in that book are mostly *data models*.

Googling will find you [this huge model for the Biomedical Research Integrated Domain Group (BRIDG)](https://bridgmodel.nci.nih.gov/files/BRIDG_Model_4.0_html/index.htm). Drilling down to the Molecular Biology sub-domain, and to the class Gene, for example, you'll see it has no behavior, nor do any of the other (conceptual) classes in this domain model.

[](https://i.stack.imgur.com/HZxpw.png)

**Are domain models in a programming language or not?**

Larman's philosophy clearly shows them as programming language-independent (*conceptual* as opposed to *software* classes), as a separate artifact from code, to tie them explicitly to the problem domain (requirements).

On the other hand, you'll find Fowler saying, ["I prefer POJO domain models."](http://www.martinfowler.com/bliki/AnemicDomainModel.html), which is pretty much saying domain models are defined in code.

Eric Evans' [DDD](https://en.wikipedia.org/wiki/Domain-driven_design) makes the assumption that an important degree of complexity in much of software development comes from the domain, and so a model of such complex domains is essential to managing the complexity. Therefore, Domain Modeling is necessary when domains are *complex*. DDD suggests using a domain modeling language that is *ubiquitous*; that is, common to the domain experts and the developers. This would imply that in at least some cases, a domain model would not be defined in a programming language.

There is a [related question](https://softwareengineering.stackexchange.com/q/213577/51948) that might shed some light (although it has generated a lot of heat). Some have criticized the question's example as being too trivial (not complex enough) for a justified domain model.

A "domain model" is simply an object which represents some discernible concept in your business domain. A "Customer", an "Order", etc. Whatever the business logic is, the tangible entities which make up that logic are the models in your domain.

Some will have lots of business logic (perhaps worth breaking up into other classes), some will have very little (or even none).

The difference between a "domain model" and any other class isn't a construct of the Java language itself, it's mainly a semantic construct of the business logic that you define.