CS 3773
Software Engineering
Lecture 6

Dr. Mark Robinson
Office: NPB 3.350

Reasons to Use a Model

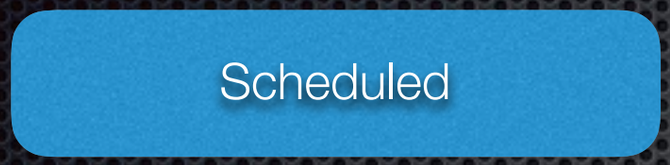
1. Helps you better understand the problem
2. The problem is so important that as many defects as possible must be found before building

But... use the models that you find helpful and appropriate. Different models provide different kinds of perspectives.

State Machine Diagrams

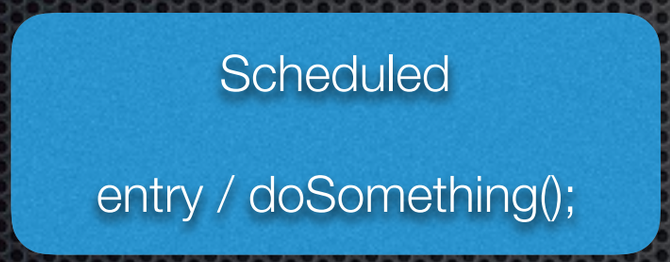
- Model dynamic behavior of
  - Use cases
  - Classes/Objects
  - Systems and subsystems
- Have states, transitions, and events
- Useful for understanding complex object behavior and state relationships

State Machine Symbols



State: a value representing a valid state of an object

State with a method: when object is in this state , continuously call doSomething();



State with an event and method: when object is in this state and the given event occurs, then call doSomething();
Events are “entry”, “exit”, or any object-specific event

State Machine Symbols

Transition: the event that changes an object's state

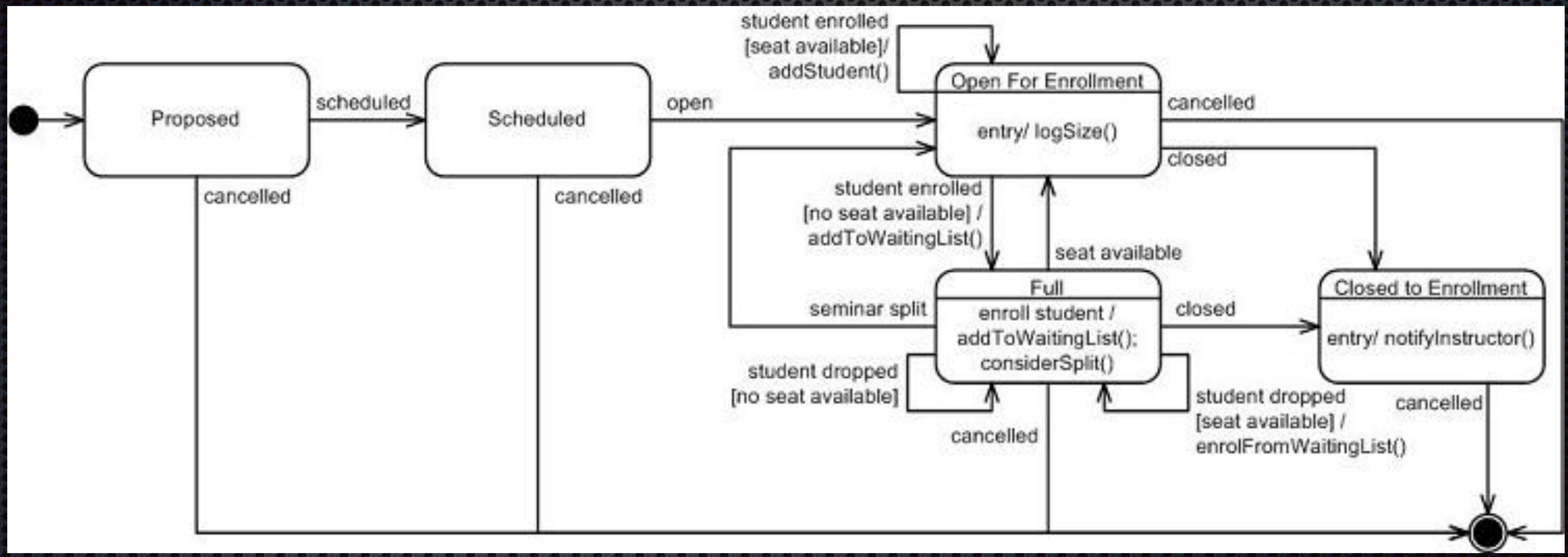
Transition with guard: transition only happens when event occurs and guard condition is true

Transition with guard and method call: transition only happens when event occurs and guard condition is true. method is called during transition.

Initial state: enter diagram here

Final state: exit diagram here

State Machine Example



Borrowed from http://agilemodeling.com/artifacts/stateMachineDiagram.htm

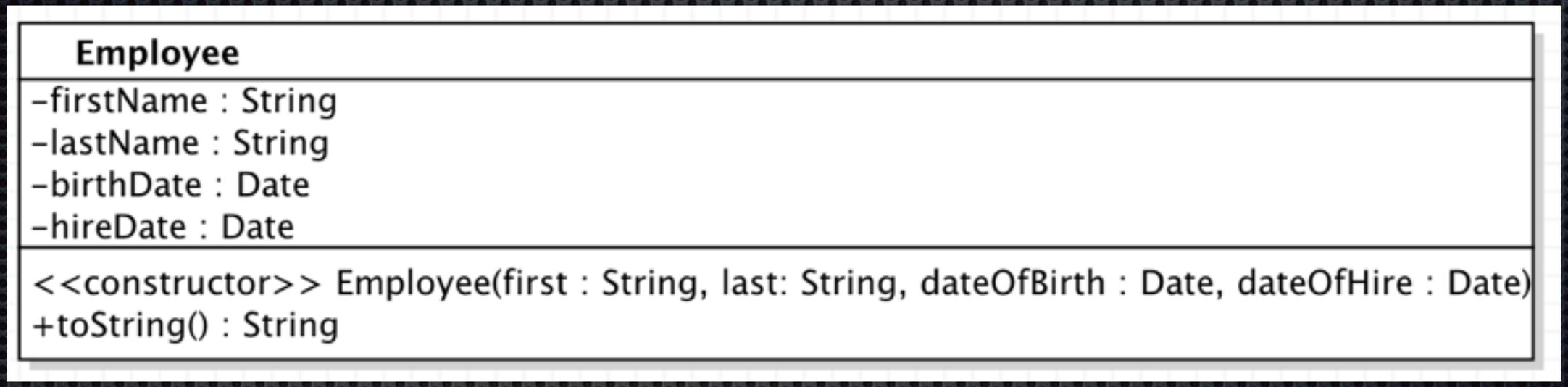
Class Diagrams

- Structural model that shows relationships between data objects
- Can include object attributes and methods
  - But quickly become too visually complicated
  - Models are **not** visual programming tools
- Each class diagram should have a specific purpose and only include the information necessary to aid the purpose

Class

- Diagramed as box with
  - Name
  - Fields (attributes) with data type
  - Methods with parameters and return type
  - Visibility indicators (public, private, static, etc.)

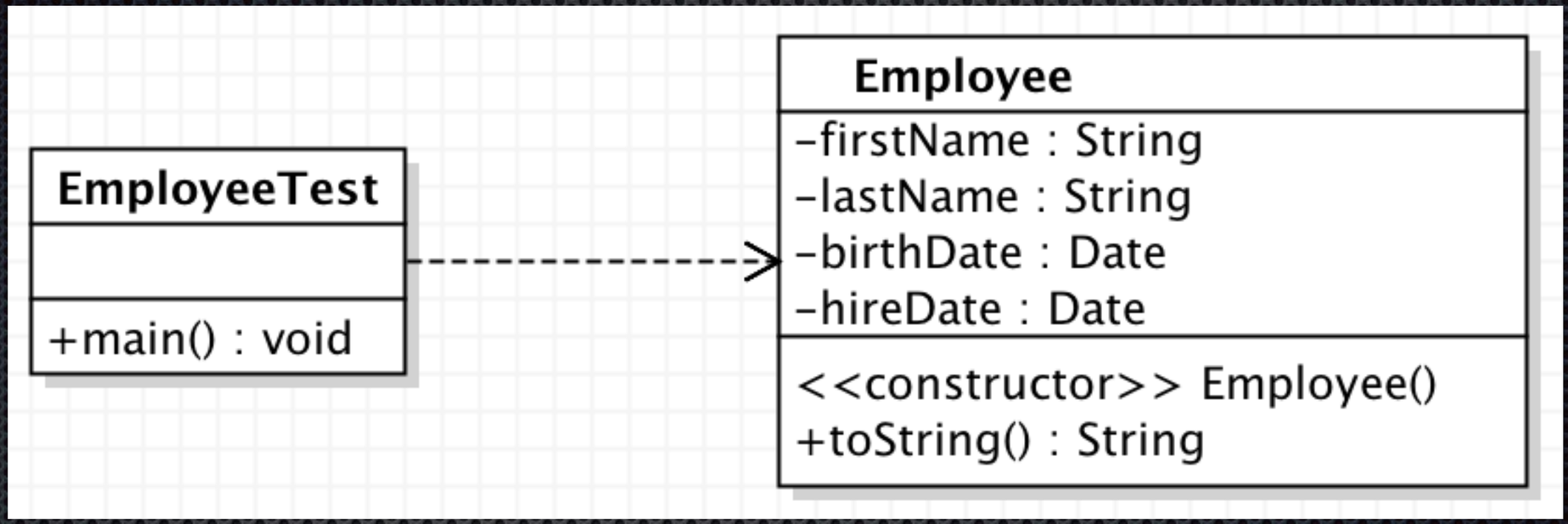
Class Example



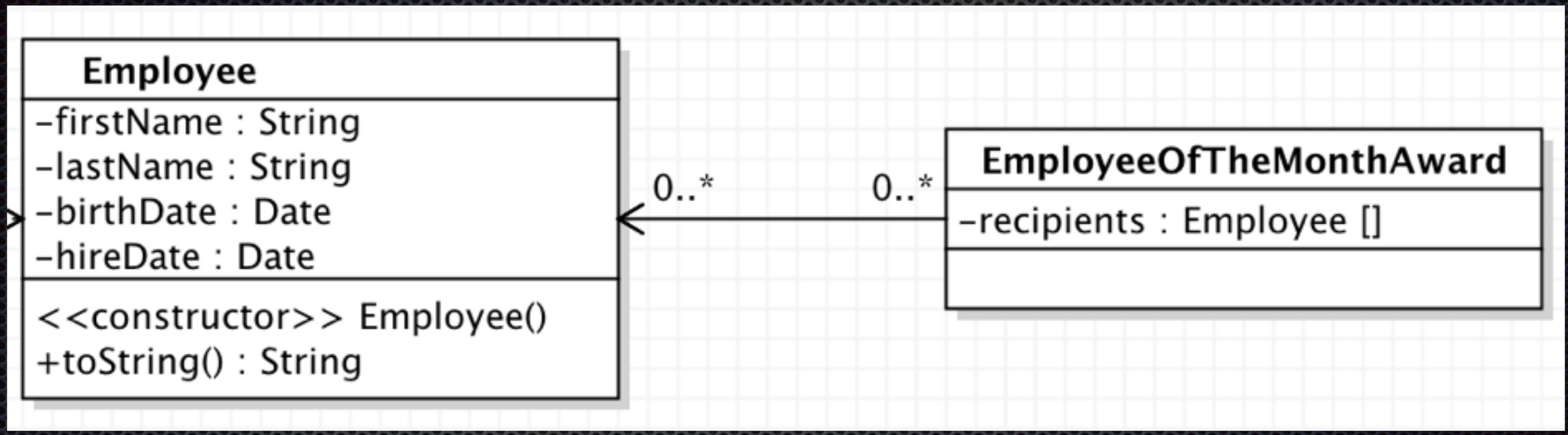
Instance and Class Relationships

- **Dependency** : c1 uses c2 but no instance variable
- **Association** (uni- and bi-directional) : knows of, independent classes (each can exist without the other)
- **Aggregation** : part-of/has-a, but contained can exist without
- **Composition** : part-of, but contained cannot exist without
- **Generalization** : inheritance/extension
- **Realization** : works like a, implements interface

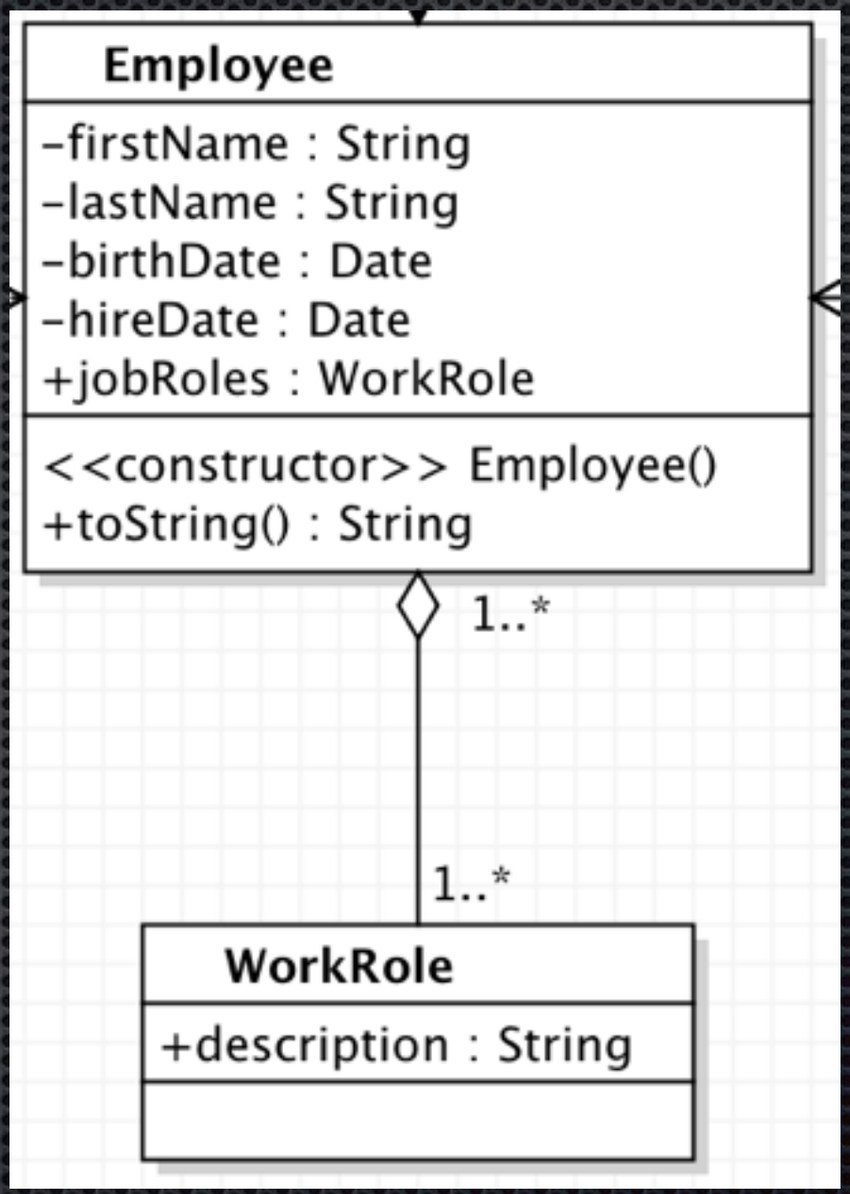
Dependency


EmployeeTest
+main() : void
Employee
-firstName : String
-lastName : String
-birthDate : Date
-hireDate : Date
<<constructor>> Employee()
+toString() : String

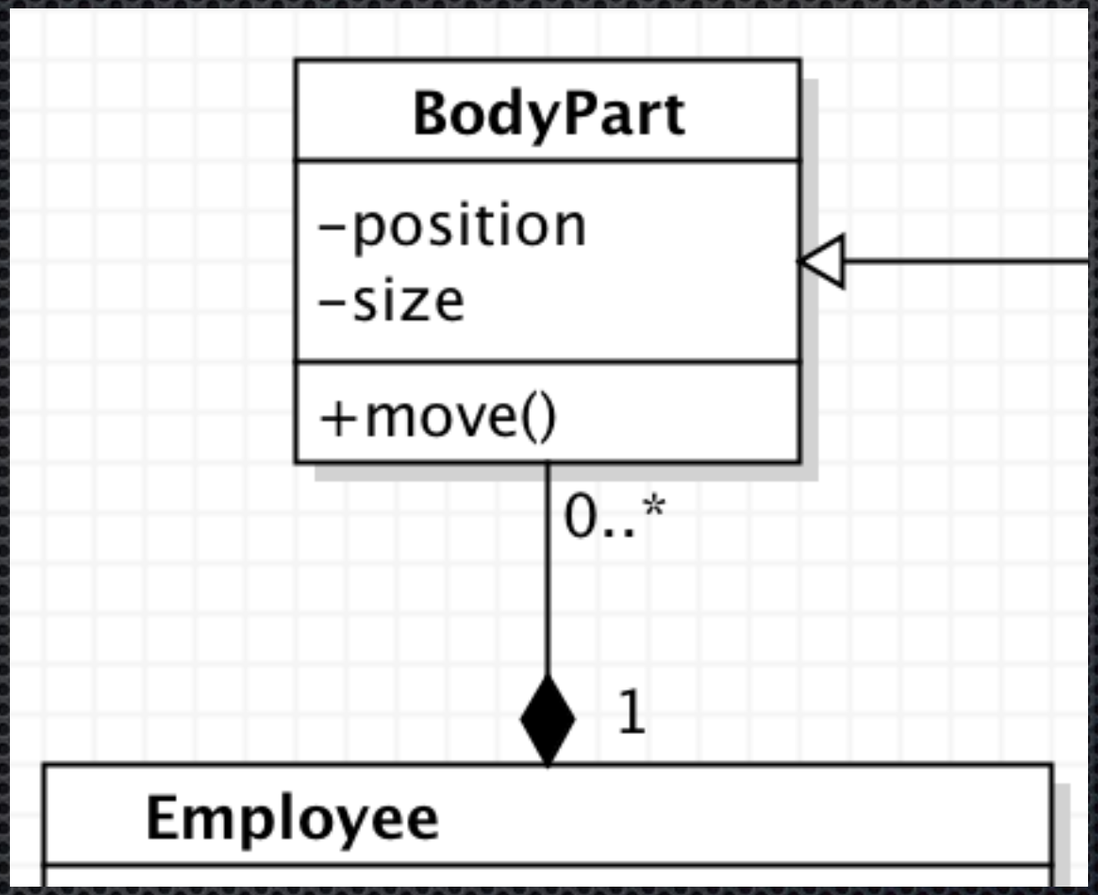
Association



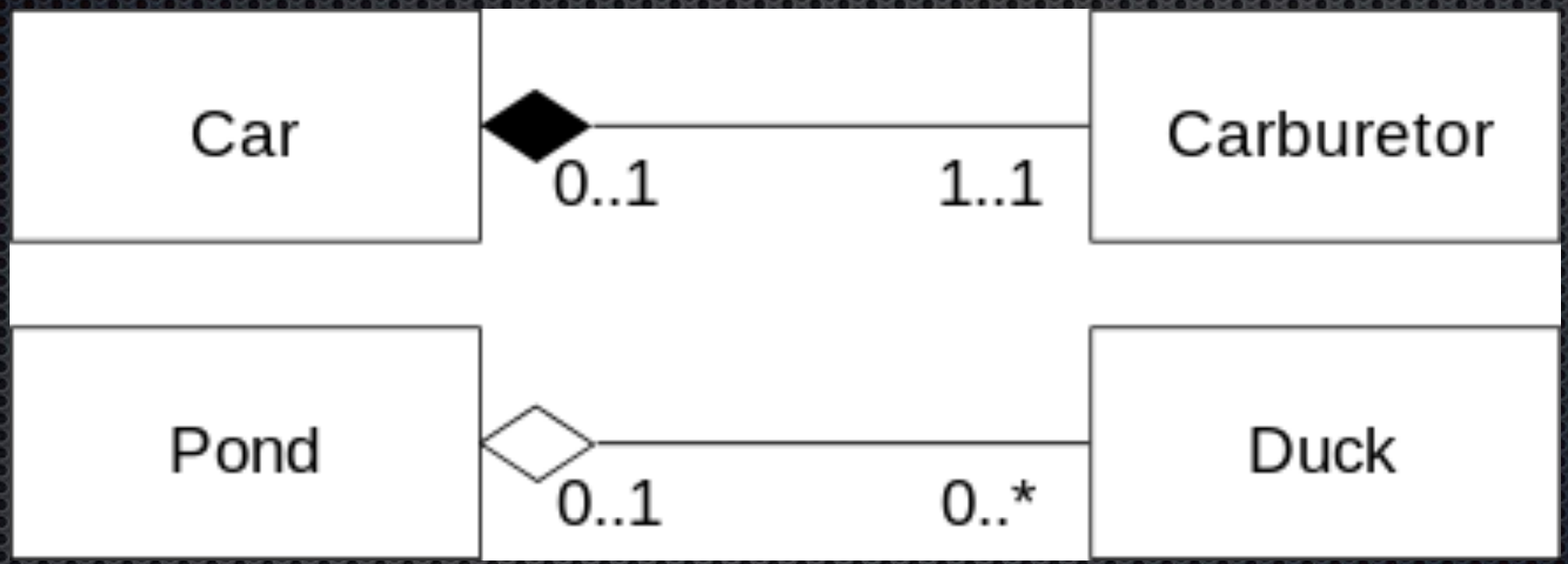
Aggregation



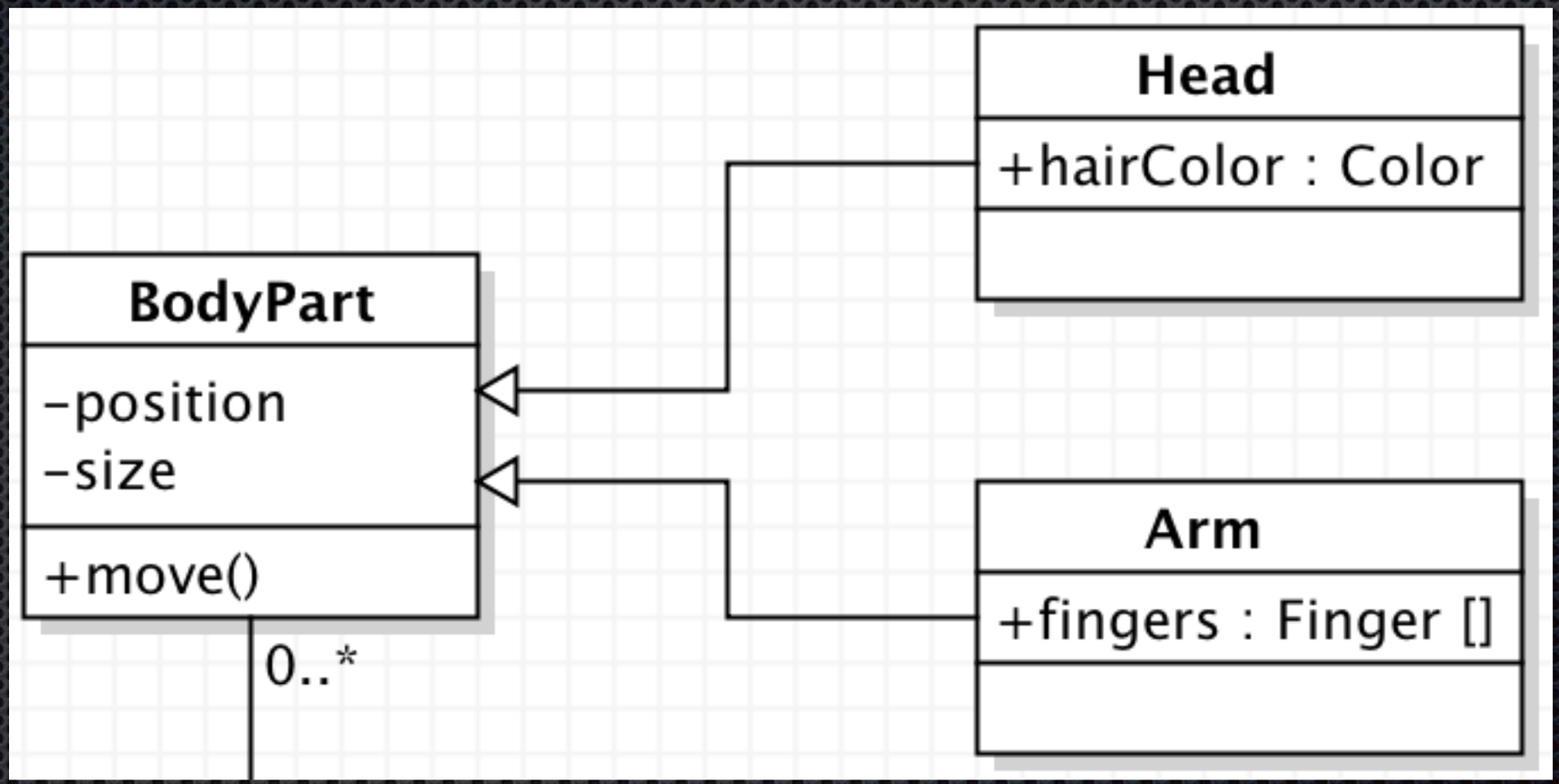
Composition



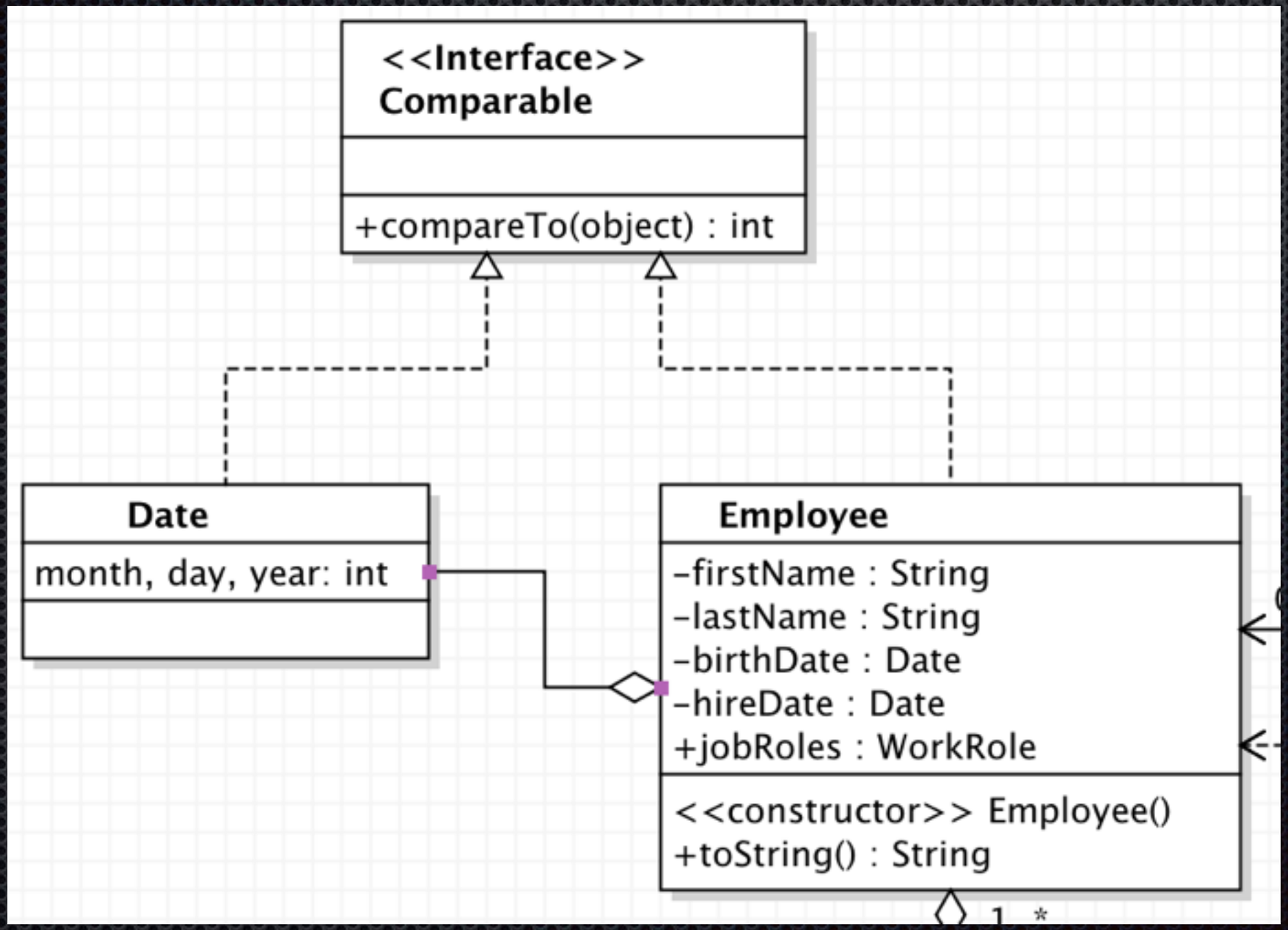
Another Example



Generalization



Realization



Multiplicity/Cardinality

- Applies to association, aggregation, composition
- **x** : exactly x # in relationship (e.g., 5)
- **x.. y** : x to y # (e.g., 0.. 5)
- **x.. *** : x to many (i.e., max is unbounded)

Relationship Takeaway

- Definitions are fuzzy
- Use something
  - Dependency : x calls y
  - Association : loose or independence
  - Aggregation : ownership (x is part of y)

Class Diagram Takeaway

- Tool for visually modeling class design
  - Useful for understanding & communicating complexity
- Loose standard (everyone has their own version)
  - Fuzzy relationship defs and notation usage
- Make it work for you (but stay in the ballpark)

Discovering Objects in Requirements

- Grammatically parse usage scenarios, use cases
- Make list of nouns/noun phrases (and synonyms)
- Categorize the nouns
  - User entities/roles, non-user **entities**, **events**, **data objects**, **attributes** of objects

Example Potential Objects (AddPropertyCoverage)

Noun	Category
Agent	User entity
Data entry person	User entity
DB Manager	User entity
Zip code DB	Non-user entity
system	The software to be built
form	possible DO
property coverage	DO
address, city, county, state, zip	Attributes of DO
insured value, coverage dates	Attributes of DO

Test Each Potential Object

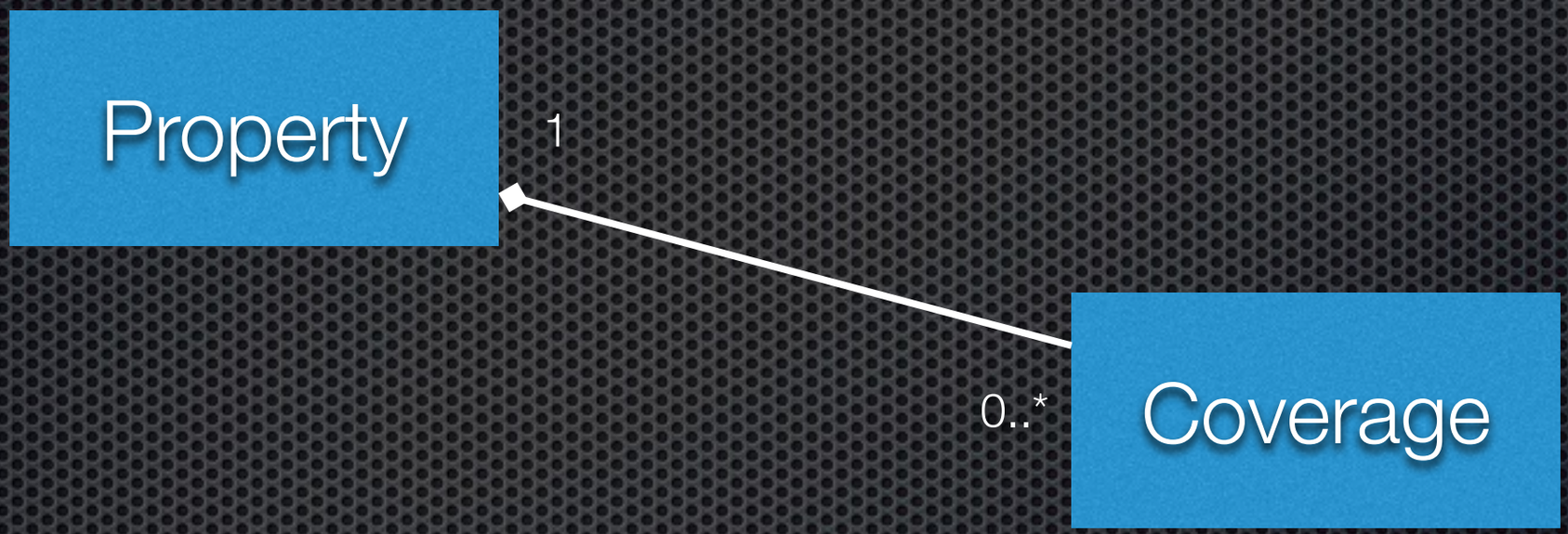
- Consider rejecting if:

1. System does not need object's information to persist
2. Object does not have identifiable attributes or operations
3. Object does not have multiple attributes
4. Object does not have common attributes or operations for instances of the object
5. Object is not a producer or consumer of information

Accepted/Rejected Objects

Noun	Category	Accept?
Agent	User entity	Yes, (user role)
Data entry person	User entity	Yes, (user role)
DB Manager	User entity	Yes, (user role)
Zip code DB	Non-user entity	Yes
form	DO	Yes
property coverage	DO	Yes
address, city, county, state, zip	Attributes	No (check #3)
insured value, coverage dates	Attributes	No (check #3)

Class Diagram to Understand/ Communicate Data Relationships



Create DO Models with Attributes and Basic Types

Property	
ID	auto
address	string
city	string
county	string
state	string
zip	string

Coverage	
insured value	real
coverage effective	date
coverage expiration	date

And Some Constraints/ Business Logic

Property		
ID	auto	
address	string	alpha/num, >0, <256
city	string	
county	string	
state	string	
zip	string	5 or 9 numeric digits

Coverage		
insured value	real	> 0, > 100,000 approve
coverage effective	date	> 60 days in past
coverage expiration	date	>= effective

Helpful to See Which Functions Modify Fields

Property			Related Functions
ID	auto		
address	string	alpha/num, >0, <256	addCoverage
city	string		addCoverage
county	string		addCoverage
state	string		addCoverage
zip	string	5 or 9 numeric digits	addCoverage

Coverage			Related Functions
insured value	real	> 0, > 100,000 approve	addCoverage
coverage effective	date	> 60 days in past	addCoverage
coverage expiration	date	>= effective	add/cancel Coverage

My Preference

- Text-based Data Object models can consolidate all of the detail for an object
- As well as applicable constraints/business logic
- Try to keep constraints in one place (DRY): easier to maintain and avoid inconsistencies from change

Class-Responsibility-Collaborator Modeling (CRC)

- A way of organizing objects (classes) and relationships
  - **Responsibility**: attribute or operation of a class
  - **Collaborator**: other class that helps with the respective responsibility
  - 3 class types: entity (things/data), boundary (user interfaces), controller (connection between entities and boundaries)

CRC Example

Class: PropertyCoverage		
Description: associates property location information with active insurance coverage		
Attributes		
address, city, county, state, zip code : text; cov. value : currency; effective, expiration : date		
Responsibility		**Collaborator**
lookup city, state, and county from zip		Zip Code DB
add coverage		DE person, agent
cancel coverage		DE person, agent
validate address, zip code		
validate coverage value and dates		
save		