En el diagrama UML podemos ver tanto las entidades, con sus atributos propios, como las relaciones y su cardinalidad. En cuanto a las entidades, comentar algunos puntos:

* Aunque barajamos con distintas opciones, como la de utilizar UUID o incluso long autoincrementales, finalmente decidimos utilizar los ObjectId como tipo en los id de las distintas clases. Al ser este el tipo propio de MongoDB / bson, hemos utilizado la etiqueta @GeneratedValue(strategy = GenerationType.IDENTITY), para generar estos identificadores.
* También tomamos la decisión de que tanto tecnologías como estado fuesen de tipo Enum. Aunque al principio nos fuese más complicado ligarlo a la base de datos que habiéndose marcado como clases con sus respectivos id, nos parecía que no tenían la suficiente entidad como para ser clases independientes. En el caso específico de tecnologías pensamos en un primer momento en embeberlo en programador, pero al aparecer también en otras clases, decidimos finalmente expresarla como una tabla a parte.

Por otro lado tenemos las relaciones y su navegabilidad. Muchas de ellas estaban más claras, marcadas por relaciones de agregación y composición. Las analizaremos de arriba a abajo.

* Login-Programador: No tendría sentido la existencia de un login sin su programador, por eso aparecerá un objeto Programador en nuestro Login. Sin embargo, como no necesitaremos hacer búsquedas del usuario desde su login, hemos decidido que esta relación sea una @OnetoOne unidireccional.
* Programador-Departamento: Nos encontramos con una relación @OneToMany o @ManyToOne entre dos de las entidades más importantes del programa. Estas estarán muy relacionadas con el resto, y serán punto de entrada de diferentes consultas, por lo que aquí tomamos la decisión de hacer esta relación bidireccional.
* Programador-Proyecto: Una @ManytoMany importante. Al igual que antes, ambas tienen mucha relevancia, aunque en este caso, para evitar grandes problemas de recursividad, y pudiendo hacer las consultas a través de otras relaciones, optamos finalmente por una unidireccional que permitiese el acceso desde programador a sus proyectos.
* Departamento-Proyecto: Para nosotros, tiene sentido guardar información de los proyectos de un departamento aunque este desaparezca, por eso, marcamos en este caso la relación como una agregación. De cara a las futuras consultas, era igual de relevante saber los proyectos de un departamento que el departamento al que pertenece un proyecto, por lo que marcamos esta relación @OneToMany o @ManyToOne como bidireccional.
* Programador-Issue: Esta relación @OneToMany la marcamos inicialmente como unidireccional. Pero al querer hacer consultas que relacionaban programadores, commits e issues, nos facilitaba la tarea el hacerla bidireccional.
* Programador-Commit: Un caso similar al anterior.
* Issue-Repositorio / Repositorio-Commit : tanto issues como commits están muy relacionados con los repositorios. En ambos casos son relaciones @OneToMany o @ManyToOne. Inicialmente también las pensamos como unidireccionales, pero la necesidad de consultas que relacionaban estas tablas con sus departamentos y programadores, nos hicieron acabar por decidir hacerlas también bidireccionales.

Como pequeño resumen. En general hemos optado por relaciones bidireccionales para poder hacer más cómodamente consultas que incluyesen información de varias tablas. Dejando como unidireccionales aquellas donde el acceso de un punto al otro está muy marcado y al revés no tendrían demasiado uso. Finalmente, comentar que para evitar problemas de stackoverflow, hemos tenido que “simplificar” los métodos toString de las diferentes clases, optando en muchos casos por imprimir solo el nombre, o elemento más característico de la clase, salvando algún caso puntual donde nos interesase mantener también el id u otros atributos.