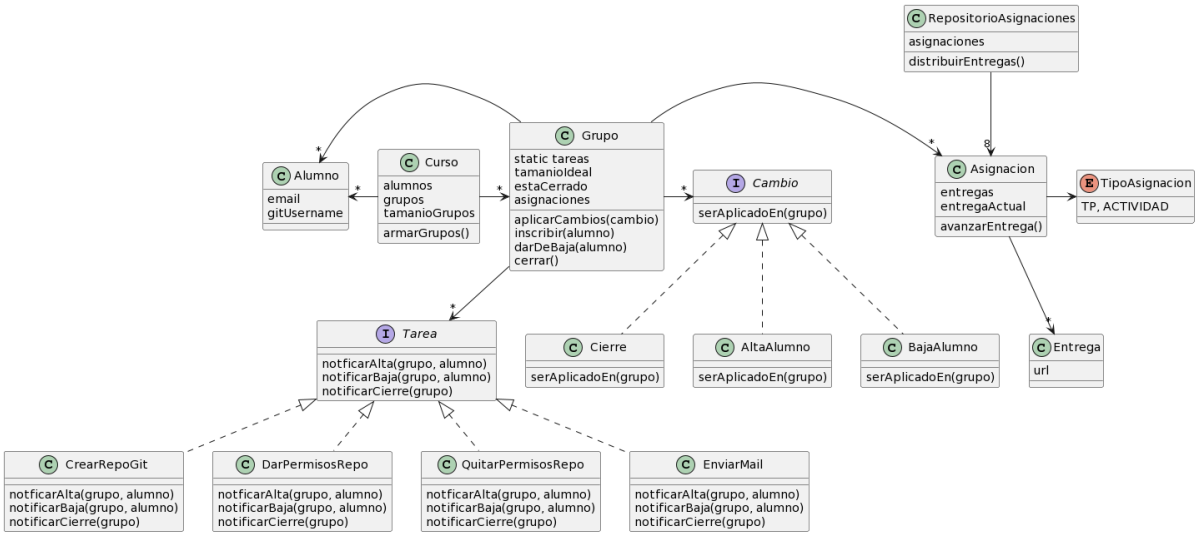
Noodle - Posible solución




```
#Curso >> generarGrupos(){
  for(1 to this.cantidadGrupos){
    this.cursos.add(new Grupo(this.tamanioGrupos))
  }
}

#Grupo >> inscribirAlumno(alumno){
  if(this.estaCerrado){
    this.posponerInscripcion(alumno)
  } else {
    this.agregarAlumno(alumno)
  }
}

#Grupo >> agregarAlumno(alumno){
    this.alumnos.add(alumno)
    this.notificarAlta(alumno)
}

#Grupo >> darBajaAlumno(alumno){
  if(this.estaCerrado){
    this.posponerBaja(alumno)
  } else {
    this.remover(alumno)
  }
}

#Grupo >> removerAlumno(alumno){
```

```
    this.alumnos.remove(alumno)
    this.notificarBaja(alumno)
}

#Grupo >> posponerInscripcion(alumno){
    this.cambiosPendientes.add(new AltaAlumno(alumno))
}

#Grupo >> posponerBaja(alumno){
    this.cambiosPendientes.add(new BajaAlumno(alumno))
}

#Grupo >> aplicarCambio(cambio){
    this.cambiosPendientes.remove(cambio)
    cambio.serAplicadoEn(this)
}

#AltaAlumno >> serAplicadoEn(grupo){
    grupo.agregarAlumno(this.alumno)
}

#BajaAlumno >> serAplicadoEn(grupo){
    grupo.removerAlumno(this.alumno)
}

#Grupo >> cerrar(){
  if(this.alumnos.size() == this.tamanioIdeal){
    this.efectivizarCierre()
  } else {
    this.posponerCierre()
  }
}

#Grupo >> posponerCierre(){
  this.cambiosPendientes.add(new Cierre())
}

#Grupo >> this.efectivizarCierre(){
  this.estaCerrado = true
  this.notificarCierre()
}

#Cierre >> serAplicadoEn(grupo){
    grupo.efectivizarCierre()
}

#Grupo >> notificarAlta(alumno){
  this.tareas.forEach(t -> t.notificarAlta(this, alumno))
}
```

```
#Grupo >> notificarBaja(alumno){
  this.tareas.forEach(t -> t.notificarBaja(this, alumno))
}

#Grupo >> notificarCierre()){
  this.tareas.forEach(t -> t.notificarCierre(this)))
}

#CrearRepo >> notificarCierre(grupo){
  new GuitabSDK().crearRepoConAccessos("repo_" +
grupo.getNombre(),grupo.getUsernames())
}

#Grupo >> getUsernames(){
  return this.alumnos.map(a -> a.getGitUsername())
}

#DarAccesosRepo >> notificarAlta(grupo, alumno){
  if(grupo.estaCerrado()){
    new GuitabSDK().darAcceso("repo_" +
grupo.getNombre(),alumno.getGitUsername())
  }
}

#QuitarAccesosRepo >> notificarAlta(grupo, alumno){
  if(grupo.estaCerrado()){
    new GuitabSDK().quitarAcceso("repo_" +
grupo.getNombre(),alumno.getGitUsername())
  }
}

#EnviarMail >> notificarAlta(grupo, alumno){
  new MailSender().send(grupo.getEmails(), "Nuevo Integrante", "Se ha sumado" +
integrante.getNombre())
}

#EnviarMail >> notificarBaja(grupo, alumno){
  new MailSender().send(grupo.getEmails(), "Baja Integrante", "Se ha retirado" +
integrante.getNombre())
}

#EnviarMail >> notificarCierre(grupo, alumno){
  new MailSender().send(grupo.getEmails(), "Se ha cerrado el grupo")
}

#RepoAsignaciones >> distribuirEntregas(){
  Curso.getinstance().getGruposCerrados().forEach(g ->
this.habilitarEntregas(g))
```

```
}

#Curso >> getGruposCerrados(){
  return this.grupos.filter(g -> g.estaCerrado())
}

//Se corre semanalmente en un main utilizando crontab
#RepoAsignaciones >> habilitarEntregas(grupo){
  this.asignaciones.forEach(a -> {
    grupo.habilitarEntrega(a.getEntregaActual())
    a.avanzarEntrega()
  })
}

#Asignacion >> avanzarEntrega(){
  this.entregaActual = this.entregas[this.entregas.indexOf(this.entregaActual) +
1]
}

#Grupo >> habilitarEntrega(entrega){
  this.entregashabilitadas.add(entrega)
  new EnviarMail().notificarEntrega(this, entrega)
}

#EnviarMail >> notificarEntrega(grupo, entrega){
  new MailSender().send(grupo.getEmails(), "Se ha liberado la entrega" +
entrega.getTitulo())
}
```

Observaciones y justificaciones

- El estado cerrado/abierto se modela como un booleano porque no hay mucha complejidad asociada más que un par de chequeos, y porque no hay ningún indicio de que a futuro estos estados puedan multiplicarse, por lo cual agregar una solución que delegue en un objeto estado se volvería bastante mas complejo innecesariamente (sobrediseño)
- Los Cambios (Alta/Baja/Cierre) se modelan como entidades que reifican el comportamiento para permitir diferirlo hasta su aprobacion. Ademas, se usa una interfaz comun (lo cual no es estrictamente necesario solo para permitir diferir las operaciones) dado que existe la posibilidad de que a futuro se desee soportar algún que otro cambio, y de esta forma se facilitaría.
- Para la aprobación del cierre de grupos, no se modela la validación de que el docente sea otro distinto dado que esto es una responsabilidad de aplicación, no de dominio. Ver Sobre el manejo de usuarios, roles y permisos
- Para las tareas que ocurren ante un cambio, utilizamos un modelado por eventos dado que se específica que esta es un area del sistema que muy variable que debe poder ser modificable *en tiempo de ejecución*
- Las tareas se almacenan en una variable de clase dado que el enunciado sugiere que son configurables de forma global para todos los grupos. De todos modos, como no se especifica con exactitud, haberlas guardado en una variable de instancia tampoco seria un error.

- Una alternativa a modelar las tareas de dar/quitar permisos y/o crear el repo, podría haber sido tener una sola tarea que haga cada una de esas cosas ante cada evento. Esto depende de la granularidad que se desea a la hora de configurar tareas (es algo que valdría la pena preguntar al usuario. En el contexto de un examen, se puede preguntar o en su defecto, asumir y anotar lo que se asume)
- Para interactuar con los sistemas externos MailSender y GuitabSDK, se reutilizan los objetos que modelan las tareas. Es interesante notar que estos objetos estan cumpliendo a la vez la responsabilidad de ser “interesados” en los cambios del grupo, y de “adaptar” la interfaz de los sistemas externos a algo más amigable. Esto tiene sentido porque de haber separado esas responsabilidades en dos objetos diferentes, se hubiera creado un pasamanos que no aportaría nada. Si bien haberlo hecho no sería un error si se justifica a nivel cohesión, sí es totalmente innecesario. De hecho, esta combinación de responsabilidades es muy común, dado que muy a menudo los interesados en un modelado por eventos necesitan interactuar con sistemas externos.
- Dado que son SDKs (librerías que encapsulan sistemas externos), podemos instanciar al MailSender y GuitabSDK sin necesidad de inyectarlos. Esto no compromete la testeabilidad dado que si quisiéramos mockear, podemos mockear a todo el objeto Tarea (por ejemplo, podríamos tener un “EnviarMailMock” que no envía mails, pero nos permite verificar que se lo haya llamado)
    - En este caso si, por ejemplo, se quisiera testear el mensaje exacto que envía por mail la tarea EnviarMail, tal vez ahí sí se justificaría separar la tarea (interesado) “EnviarMail” del “adaptador” “EnviadorDeMails” que instanciaría y llamaría al MailSender, pudiendo mockear a este último
- Se modela al Curso como singleton por simplicidad, dado que no se especifica si esta aplicación existe en el contexto de un sistema general para varios cursos o es un sistema del cual se ejecuta una instancia por curso. Alternativamente, se podría haber pensado en un RepositorioCursos que conozca todos los cursos y agregar un nivel más de forEach al distribuir las asignaciones.
- Nótese que el metodo EnviarMail#`notificarEntrega` es propio sólo de esta clase y no de la interfaz Tarea. Eso es porque en este caso no esta representando un evento sino solamente comportamiento que debe proveer esa clase en su calidad de “adaptador” del sistema externo “MailSender”
- Finalmente, una observación interesante es que si se analiza, se notará que hay una correspondencia entre los “cambios” y los eventos que se informan a las tareas (AltaAlumno -> notificarAltaAlumno, Cierre -> notificarCierre, BajaAlumno -> notificarBajaAlumno). Esto implica (como los eventos se ven representados en los métodos de la interfaz “Tarea”), que probablemente la adición/remoción de cambios a futuro implicaría modificar esta interfaz. Esto podría verse como un problema, sin embarazo el enunciado comenta que la implementación de nuevos “cambios” es poco probable y si ocurre no se espera que ocurra demasiado. Además, no se necesita soportar que varíen en tiempo de ejecución. Poe ello, no es un problema modelar los eventos de esta forma.

Sobre el manejo de usuarios, roles y permisos

El manejo de usuarios, roles y permisos no es una responsabilidad intrínseca de las clases de dominio sino que es una responsabilidad de aplicación. Solo manejaremos cuestiones de permisos de usuarios en el dominio cuando estos sean partes propias del dominio (por ejemplo, porque nuestro dominio fuera, justamente, un sistema de manejo de permisos). Las responsabilidades de aplicación son aquellas responsabilidades relacionadas a cómo acceder a los objetos de nuestro dominio y manejarlos para ejecutar los casos de uso. Están intimamente relacionadas, también, con los roles que interactúan con el sistema. Estas responsabilidades están en muchos casos intimamente relacionadas con las de interfaz de usuario (aunque no siempre) y tienen mayor dependencia sobre la arquitectura de nuestra aplicación. En lo que respecta a la materia, sobre ellas estaremos trabajando en la segunda parte del año. Para ejemplificar, la responsabilidad de asignar “cambios pendiente de aprobacion” a distintos docentes, mostrarlos en pantalla o validar si tal o cual usuario puede aprobar un cambio o no, debemos saber si nuestra aplicación está corriendo en un servidor web común a todos los usuarios, en cuyo caso debería haber una noción de asignación y validacion como la antes mencionada, o si la aplicación se ejecuta en un teléfono celular, donde el rol del usuario actual es siempre conocido y alcanzaría a lo sumo con solo saber qué usuario fue el que intentó cerrar el grupo para luego evitar mostrarle ese grupo en su lista de cambios por aprobar. Es decir, no es que esta responsabilidad sea poco importante o que no pueda impactar en como modelamos el dominio, sino que involucra aspectos que exceden al dominio en sí y, por ello, no tenemos suficiente información para modelarlo en esta instancia (de la materia, o del diseño de la aplicación) y posponemos su modelado.
Por último, sí cabe destacar que no es incorrecto tener objetos del tipo “usuario” u objetos que puedan tener el mismo nombre que algún tipo de rol del sistema, en tanto y en cuanto esos objetos cumplan responsabilidades de dominio y no de aplicación (Es decir, que no sean objetos que “manejan” a otros objetos e “imitan” lo que haría una persona en la vida real)