Testing report

Diseño y pruebas II

Sprint 4

Versión	Fecha	Descripción
1.0	15/05/2024	Creación del documento.
1.1	20/05/2024	Adición de la estructura básica
1.2	26/04/2024	Adición de la información referente al capítulo de testeo funcional
1.3	27/04/2024	Adición de la información del capítulo del análisis del rendimiento de los tests y revisión final

https://github.com/DP2-c1-028/Acme-SF-D04

Fecha 15/05/2024

Preparado por:
Benito Hidalgo, Daniel (C1.028) (danbenhid@alum.us.es)

Índice

Introducción

El objetivo de este documento es mostrar el proceso de testeo desarrollado para el proyecto, en este se indagará en la metodología usada para la realización de los test, las herramientas usadas para la creación de estos y el análisis de los resultados tras la ejecución de los tests.

El documento sigue la estructura recomendada por el profesorado, en el que se diferencian 2 grandes partes: la realización del testeo formal, donde se explicarán los diferentes casos que han sido testeados, la cobertura de estos con respecto al código realizado y la justificación de esta y la efectividad de la realización de los tests para encontrar bugs.
La otra gran mitad del documento se basa en los análisis de la actuación de los tests, mostrando diferentes métricas que apoyen el intervalo de confianza alcanzado, así como el análisis del rendimiento de los tests en diferentes equipos y las conclusiones sacadas de este proceso.

1. TESTEO FUNCIONAL

1.1 testeo funcional sobre la entidad contract

Primero nos centraremos en las funcionalidades relacionadas con la primera entidad, Contract:

Funcionalidad:	Creación de contratos
Clase:	**Descripción del testeo realizado**
create.safe	Para este caso se ha seguido la metodología propuesta en la que se envía un formulario vacío, y tras esto se van probando los distintos casos negativos atributo a atributo y uno por cada vez. Una vez probado todo, se han enviado varios formularios correctos probando los distintos valores permitidos teniendo en cuenta límites y caracteres especiales (intentos de hackeo, charsets distintos o caracteres especiales).
create.hack	Para este caso se ha enviado la url perteneciente a la creación sin haber iniciado sesión, tras esto, se ha iniciado sesión con un rol distinto y se ha vuelto a enviar la petición para comprobar que no deja acceder a esta lanzando un error 500 “ access is not authorized”.
bugs encontrados gracias al testeo	No se han encontrado bugs durante la realización de estos tests.

Se puede ver el código y la cobertura alcanzada abajo:

```
@Service
public class ClientContractCreateService extends AbstractService<Client, Contract> {

    @Autowired
    private ClientContractRepository        repository;

    @Autowired
    private SystemConfigurationRepository   sysConfigRepository;


    @Override
    public void authorise() {
        super.getResponse().setAuthorised(true);

    }

    @Override
    public void load() {
        Contract contract;

        contract = new Contract();
        Integer clientId = super.getRequest().getPrincipal().getActiveRoleId();
        Client client = this.repository.findClientById(clientId);

        contract.setClient(client);
        contract.setDraftMode(true);
        super.getBuffer().addData(contract);

    }

    @Override
    public void bind(final Contract contract) {
        assert contract != null;

        super.bind(contract, "code", "instantiationMoment", "providerName", "customerName", "goals", "budget",
"project");

    }

    @Override
    public void validate(final Contract contract) {

        assert contract != null;

        if (!super.getBuffer().getErrors().hasErrors("budget") && contract.getProject() != null &&
this.sysConfigRepository.existsCurrency(contract.getBudget().getCurrency())) {
            Project referencedProject = contract.getProject();
            Double projectCost =
this.sysConfigRepository.convertToUsd(referencedProject.getCost()).getAmount();
            Double budgetUSD = this.sysConfigRepository.convertToUsd(contract.getBudget()).getAmount();

            super.state(projectCost >= budgetUSD, "budget", "client.contract.form.error.budget");
        }

        if (!super.getBuffer().getErrors().hasErrors("code")) {
            Contract contractWithCode = this.repository.findContractByCode(contract.getCode());
            super.state(contractWithCode == null, "code", "client.contract.form.error.code");
        }

        if (!super.getBuffer().getErrors().hasErrors("project"))
            super.state(!contract.getProject().isDraftMode(), "project", "client.contract.form.error.project");

        if (!super.getBuffer().getErrors().hasErrors("budget") &&
this.sysConfigRepository.existsCurrency(contract.getBudget().getCurrency())) {
            boolean validBudget = contract.getBudget().getAmount() >= 0. &&
this.sysConfigRepository.convertToUsd(contract.getBudget()).getAmount() <= 1000000.0;
            super.state(validBudget, "budget", "client.contract.form.error.budget-negative");
        }

        if (!super.getBuffer().getErrors().hasErrors("budget") && contract.getBudget() != null) {
```

```
            String currency = contract.getBudget().getCurrency();
            boolean existsCurrency = this.sysConfigRepository.existsCurrency(currency);
            super.state(existsCurrency, "budget", "client.contract.form.error.currency");
        }

        if (!super.getBuffer().getErrors().hasErrors("instantiationMoment")) {

            Date contractDate = contract.getInstantiationMoment();
            Date minimunDate = MomentHelper.parse("2000-01-01 00:00", "yyyy-MM-dd HH:mm");

            Boolean isAfter = contractDate.after(minimunDate);
            super.state(isAfter, "instantiationMoment", "client.contract.form.error.instantiationMoment");
        }

    }

    @Override
    public void perform(final Contract contract) {
        assert contract != null;
        contract.setId(0);
        this.repository.save(contract);
    }

    @Override
    public void unbind(final Contract contract) {
        assert contract != null;

        Dataset dataset;
        String projectCode;

        projectCode = contract.getProject() != null ? contract.getProject().getCode() : null;

        Collection<Project> projects = this.repository.findlAllPublishedProjects();

        SelectChoices options;

        Project project = contract.getProject() != null ? contract.getProject() : null;

        options = SelectChoices.from(projects, "code", project);

        dataset = super.unbind(contract, "code", "project", "providerName", "customerName",
"instantiationMoment", "budget", "goals", "draftMode");

        dataset.put("project", projectCode);
        dataset.put("projects", options);

        super.getResponse().addData(dataset);
    }
}
```

Como se puede ver hay varios fragmentos con una cobertura deficiente, siendo la razón de esta la siguiente:

Fragmento	Justificación de cobertura
`assert contract != null;`	Esta estructura es heredada de los proyectos de ejemplo y supone una situación que no es posible controlar por nosotros por que pertenece al funcionamiento del framework.
`if (!super.getBuffer().getErrors().hasErrors("budget") && contract.getBudget() != null)`	No es posible pasar todos los estados, en este caso que un presupuesto sea nulo y el campo del atributo no tenga errores.
`super.state(!contract.getProject().isDraftMode(), "project",`	Este fragmento supone una validación realizada con el fin de evitar un posible hackeo en el

`"client.contract.form.error.project");`	campo de proyecto, el usuario común nunca va a encontrar posibilidad de llegar a este estado usando el sistema de forma correcta.

Funcionalidad:	Listado de contratos
Clase:	**Descripción del testeo realizado**
list.safe	Para este caso se ha accedido como el rol indicado (cliente) y se ha realizado la petición para mostrar el listado de contratos asociados, esto se ha llevado a cabo con varios usuarios de rol cliente.
list.hack	Para este caso se ha enviado la url perteneciente a la acción de listar contratos sin haber iniciado sesión, tras esto, se ha iniciado sesión con un rol distinto(administrator) y se ha vuelto a enviar la petición para comprobar que no deja acceder a esta lanzando un error 500 “ access is not authorized”.
bugs encontrados gracias al testeo	No se han encontrado bugs durante la realización de estos tests.

Se puede ver el código realizado abajo:

```
@Service
public class ClientContractListService extends AbstractService<Client, Contract> {

    // Internal state ---------------------------------------------------------

    @Autowired
    private ClientContractRepository repository;

    // AbstractService interface ----------------------------------------------


    @Override
    public void authorise() {
        super.getResponse().setAuthorised(true);

    }

    @Override
    public void load() {

        Collection<Contract> contracts;
        int clientId;

        clientId = super.getRequest().getPrincipal().getActiveRoleId();
        contracts = this.repository.findContractsByClientId(clientId);

        super.getBuffer().addData(contracts);
    }

    @Override
```

```
    public void unbind(final Contract contract) {

        assert contract != null;
        Dataset dataset;
        String projectCode = this.repository.findProjectById(contract.getProject().getId()).getCode();

        dataset = super.unbind(contract, "code", "project", "draftMode", "providerName", "customerName",
"instantiationMoment", "budget", "goals");

        dataset.put("project", projectCode);
        super.getResponse().addData(dataset);
    }
}
```

Como se puede ver hay un fragmento con una cobertura deficiente, siendo la razón de esta la siguiente:

Fragmento	Justificación de cobertura
`assert contract != null;`	Esta estructura es heredada de los proyectos de ejemplo y supone una situación que no es posible controlar por nosotros por que pertenece al funcionamiento del framework.

Funcionalidad:	Mostrar detalles de contrato
Clase:	**Descripción del testeo realizado**
show.safe	Para este caso se ha accedido como el rol indicado (cliente) y se ha realizado una petición para mostrar los detalles de un contrato específico, esto se ha llevado a cabo con varios usuarios de rol cliente y mostrando los detalles de varios contratos por cada uno.
show.hack	Para este caso se ha enviado una url solicitando los detalles de un contrato sin haber iniciado sesión, tras esto, se ha iniciado sesión con un rol distinto(administrator) y se ha vuelto a enviar la petición para comprobar que no deja acceder a esta, por último, se ha iniciado sesión con un usuario del rol correcto y se ha realizado una petición dirigida a un contrato que no le pertenece, en todos los casos se lanza un error 500 “access is not authorized”.
bugs encontrados gracias al testeo	No se han encontrado bugs durante la realización de estos tests.

Se puede ver el código realizado junto a la cobertura abajo:

```java
@Service
public class ClientContractShowService extends AbstractService<Client, Contract> {

    // Internal state ---------------------------------------------------------
    @Autowired
    private ClientContractRepository repository;

    // AbstractService interface ----------------------------------------------
    @Override
    public void authorise() {

        int contractId;
        Contract contract;
        int clientId;
        boolean isValid;

        contractId = super.getRequest().getData("id", int.class);
        contract = this.repository.findContractById(contractId);
        clientId = super.getRequest().getPrincipal().getActiveRoleId();

        isValid = clientId == contract.getClient().getId();

        super.getResponse().setAuthorised(isValid);
    }

    @Override
    public void load() {

        Contract contract;
        int contractId;

        contractId = super.getRequest().getData("id", int.class);
        contract = this.repository.findContractById(contractId);

        super.getBuffer().addData(contract);
    }

    @Override
    public void unbind(final Contract contract) {

        assert contract != null;

        Dataset dataset;
        String projectCode = this.repository.findProjectById(contract.getProject().getId()).getTitle();

        Collection<Project> projects = this.repository.findlAllPublishedProjects();
        SelectChoices options;

        options = SelectChoices.from(projects, "code",
this.repository.findProjectById(contract.getProject().getId()));

        dataset = super.unbind(contract, "code", "project", "draftMode", "providerName", "customerName",
"instantiationMoment", "budget", "goals");

        dataset.put("project", projectCode);
        dataset.put("projects", options);

        super.getResponse().addData(dataset);
    }

}
```

Como se puede ver hay un fragmento con una cobertura deficiente, siendo la razón de esta la siguiente:

Fragmento	Justificación de cobertura
`assert contract != null;`	Esta estructura es heredada de los proyectos de ejemplo y supone una situación que no es posible controlar por nosotros por que pertenece

	al funcionamiento del framework.

Funcionalidad:	Actualización de contratos
Clase:	**Descripción del testeo realizado**
update.safe	Se ha seguido la metodología propuesta en clase, en la que se envía un formulario vacío y tras eso se prueban los casos negativos atributo a atributo. Una vez realizada esta parte, se procede al envío de formularios de actualización con datos válidos entre los límites establecidos para cada atributo. También se prueban valores especiales para el texto (intentos de hackeo, charsets distintos o caracteres especiales).
update.hack	En este caso se ha realizado una modificación en los formularios enviados con el fin de alterar la id del contrato que se estaba modificando, probando que un usuario con el rol correcto no pudiera alterar información de contratos que no son suyos, o que no deberían poder modificarse (contratos ya publicados). En todas las ocasiones se lanza un error 500 “access is not authorized”.
bugs encontrados gracias al testeo	Debido a que esta funcionalidad fue revisada a conciencia antes de desarrollar los tests, no se han encontrado bugs durante la realización del testeo.

Se puede ver el código realizado abajo:

```
@Service
public class ClientContractUpdateService extends AbstractService<Client, Contract> {

    // Internal state ---------------------------------------------------------

    @Autowired
    private ClientContractRepository        repository;

    @Autowired
    private SystemConfigurationRepository   sysConfigRepository;

    // AbstractService interface ----------------------------------------------


    @Override
    public void authorise() {
        int contractId;
        Contract contract;
        int clientId;
        boolean isValid;

        contractId = super.getRequest().getData("id", int.class);
```

```
        contract = this.repository.findContractById(contractId);
        clientId = super.getRequest().getPrincipal().getActiveRoleId();

        isValid = clientId == contract.getClient().getId() && contract.isDraftMode();

        super.getResponse().setAuthorised(isValid);
    }

    @Override
    public void load() {
        Contract contract;
        Integer contractId;

        contractId = super.getRequest().getData("id", int.class);
        contract = this.repository.findContractById(contractId);

        super.getBuffer().addData(contract);
    }

    @Override
    public void bind(final Contract contract) {
        assert contract != null;

        Integer managerId = super.getRequest().getPrincipal().getActiveRoleId();
        Client client = this.repository.findClientById(managerId);

        contract.setClient(client);
        super.bind(contract, "code", "project", "providerName", "customerName", "instantiationMoment",
"budget", "goals");

    }

    @Override
    public void validate(final Contract contract) {
        assert contract != null;

        if (!super.getBuffer().getErrors().hasErrors("budget") && contract.getProject() != null &&
this.sysConfigRepository.existsCurrency(contract.getBudget().getCurrency())) {
            Project referencedProject = contract.getProject();
            Double projectCost =
this.sysConfigRepository.convertToUsd(referencedProject.getCost()).getAmount();
            Double budgetUSD = this.sysConfigRepository.convertToUsd(contract.getBudget()).getAmount();

            super.state(projectCost >= budgetUSD, "budget", "client.contract.form.error.budget");
        }

        if (!super.getBuffer().getErrors().hasErrors("code")) {
            Contract contractWithCode = this.repository.findContractByCode(contract.getCode());

            if (contractWithCode != null)
                super.state(contractWithCode.getId() == contract.getId(), "code",
"client.contract.form.error.code");
        }

        if (!super.getBuffer().getErrors().hasErrors("project"))
            super.state(!contract.getProject().isDraftMode(), "project", "client.contract.form.error.project");

        if (!super.getBuffer().getErrors().hasErrors("budget") &&
this.sysConfigRepository.existsCurrency(contract.getBudget().getCurrency())) {
            boolean validBudget = contract.getBudget().getAmount() >= 0. &&
this.sysConfigRepository.convertToUsd(contract.getBudget()).getAmount() <= 1000000.0;
            super.state(validBudget, "budget", "client.contract.form.error.budget-negative");
        }

        if (!super.getBuffer().getErrors().hasErrors("budget") && contract.getBudget() != null) {
            String currency = contract.getBudget().getCurrency();
            boolean existsCurrency = this.sysConfigRepository.existsCurrency(currency);
            super.state(existsCurrency, "budget", "client.contract.form.error.currency");
        }

        if (!super.getBuffer().getErrors().hasErrors("instantiationMoment")) {

            Date contractDate = contract.getInstantiationMoment();
            Date minimunDate = MomentHelper.parse("2000-01-01 00:00", "yyyy-MM-dd HH:mm");

            Boolean isAfter = contractDate.after(minimunDate);
```

```
            super.state(isAfter, "instantiationMoment", "client.contract.form.error.instantiationMoment");
        }
    }

    @Override
    public void perform(final Contract contract) {
        assert contract != null;

        this.repository.save(contract);
    }

    @Override
    public void unbind(final Contract contract) {
        assert contract != null;

        Dataset dataset;
        String projectCode;

        projectCode = contract.getProject() != null ? contract.getProject().getCode() : null;

        Collection<Project> projects = this.repository.findlAllPublishedProjects();

        SelectChoices options;

        Project project = contract.getProject() != null ? contract.getProject() : (Project)
projects.toArray()[0];

        options = SelectChoices.from(projects, "code", project);

        dataset = super.unbind(contract, "code", "project", "providerName", "customerName",
"instantiationMoment", "budget", "goals", "draftMode");

        dataset.put("project", projectCode);
        dataset.put("projects", options);

        super.getResponse().addData(dataset);
    }
}
```

Como se puede ver hay varios fragmentos con una cobertura baja, la razón de esta situación es la siguiente:

Fragmento	Justificación de cobertura
`assert contract != null;`	Esta estructura es heredada de los proyectos de ejemplo y supone una situación que no es posible controlar por nosotros por que pertenece al funcionamiento del framework.
`if (!super.getBuffer().getErrors().hasErrors("budget") && contract.getBudget() != null)`	No es posible pasar todos los estados, en este caso que un presupuesto sea nulo y el campo del atributo no tenga errores.
`super.state(!contract.getProject().isDraftMode(), "project", "client.contract.form.error.project");`	Este fragmento supone una validación realizada con el fin de evitar un posible hackeo en el campo de proyecto, el usuario común nunca va a encontrar posibilidad de llegar a este estado usando el sistema de forma correcta.

Funcionalidad:	Publicación de contratos
Clase:	**Descripción del testeo realizado**
publish.safe	Para este caso se ha seguido la metodología propuesta en la que se envía un formulario vacío, y tras esto se van probando los distintos casos negativos atributo a atributo y uno por cada vez. Una vez probado todo, se han enviado varios formularios de publicación correctos probando los distintos valores permitidos teniendo en cuenta límites y caracteres especiales (intentos de hackeo, charsets distintos o caracteres especiales).
publish.hack	En este caso se ha realizado una modificación en los formularios enviados con el fin cambiar la id del contrato al que se hace referencia en el formulario, probando que un usuario con el rol correcto no pudiera publicar contratos que no son suyos, o que no deberían poder publicarse porque ya lo están lanzando un error 500 “access is not authorized”.
bugs encontrados gracias al testeo	Gracias a una revisión a conciencia de esta funcionalidad, no se han encontrado bugs durante el desarrollo de los tests, corroborando la efectividad de la revisión realizada.

El código realizado y la cobertura de este se puede consultar abajo:

```
@Service
public class ClientContractPublishService extends AbstractService<Client, Contract> {

    // Internal state ---------------------------------------------------------
    @Autowired
    private ClientContractRepository        repository;

    @Autowired
    private SystemConfigurationRepository   sysConfigRepository;

    // AbstractService interface ----------------------------------------------


    @Override
    public void authorise() {

        int contractId;
        Contract contract;
        int clientId;
        boolean isValid;

        contractId = super.getRequest().getData("id", int.class);
        contract = this.repository.findContractById(contractId);
        clientId = super.getRequest().getPrincipal().getActiveRoleId();

        isValid = clientId == contract.getClient().getId() && contract.isDraftMode();

        super.getResponse().setAuthorised(isValid);
    }

    @Override
    public void bind(final Contract contract) {
        assert contract != null;
```

```
        super.bind(contract, "code", "project", "draftMode", "providerName", "customerName",
"instantiationMoment", "budget", "goals");
    }

    @Override
    public void validate(final Contract contract) {
        assert contract != null;

        if (!super.getBuffer().getErrors().hasErrors("budget") && contract.getProject() != null &&
this.sysConfigRepository.existsCurrency(contract.getBudget().getCurrency())) {

            int projectId = contract.getProject().getId();
            Collection<Contract> contracts = this.repository.findPublishedContractsByProjectId(projectId);

            if (!contracts.isEmpty()) {

                Double totalBudgetUsd = contracts.stream().mapToDouble(u ->
this.sysConfigRepository.convertToUsd(u.getBudget()).getAmount()).sum();
                Double projectCostUsd =
this.sysConfigRepository.convertToUsd(contract.getProject().getCost()).getAmount();
                double afterPublishingTotalCostUsd = totalBudgetUsd +
this.sysConfigRepository.convertToUsd(contract.getBudget()).getAmount();

                super.state(afterPublishingTotalCostUsd <= projectCostUsd, "*",
"client.contract.form.error.publishError");
            }
        }

        if (!super.getBuffer().getErrors().hasErrors("budget") && contract.getProject() != null &&
this.sysConfigRepository.existsCurrency(contract.getBudget().getCurrency())) {
            Project referencedProject = contract.getProject();
            Double projectCost =
this.sysConfigRepository.convertToUsd(referencedProject.getCost()).getAmount();
            Double budgetUSD = this.sysConfigRepository.convertToUsd(contract.getBudget()).getAmount();

            super.state(projectCost >= budgetUSD, "budget", "client.contract.form.error.budget");
        }

        if (!super.getBuffer().getErrors().hasErrors("code")) {

            Contract contractWithCode = this.repository.findContractByCode(contract.getCode());

            if (contractWithCode != null)
                super.state(contractWithCode.getId() == contract.getId(), "code",
"client.contract.form.error.code");
        }

        if (!super.getBuffer().getErrors().hasErrors("project"))
            super.state(!contract.getProject().isDraftMode(), "project", "client.contract.form.error.project");

        if (!super.getBuffer().getErrors().hasErrors("budget") &&
this.sysConfigRepository.existsCurrency(contract.getBudget().getCurrency())) {
            boolean validBudget = contract.getBudget().getAmount() >= 0. &&
this.sysConfigRepository.convertToUsd(contract.getBudget()).getAmount() <= 1000000.0;
            super.state(validBudget, "budget", "client.contract.form.error.budget-negative");
        }

        if (!super.getBuffer().getErrors().hasErrors("budget") && contract.getBudget() != null) {
            String currency = contract.getBudget().getCurrency();
            boolean existsCurrency = this.sysConfigRepository.existsCurrency(currency);
            super.state(existsCurrency, "budget", "client.contract.form.error.currency");
        }

        if (!super.getBuffer().getErrors().hasErrors("instantiationMoment")) {

            Date contractDate = contract.getInstantiationMoment();
            Date minimunDate = MomentHelper.parse("2000-01-01 00:00", "yyyy-MM-dd HH:mm");

            Boolean isAfter = contractDate.after(minimunDate);
            super.state(isAfter, "instantiationMoment", "client.contract.form.error.instantiationMoment");
        }

    }

    @Override
    public void load() {
```

```
        Contract contract;
        int contractId;

        contractId = super.getRequest().getData("id", int.class);
        contract = this.repository.findContractById(contractId);

        super.getBuffer().addData(contract);
    }

    @Override
    public void perform(final Contract contract) {
        assert contract != null;

        contract.setDraftMode(false);
        this.repository.save(contract);
    }

    @Override
    public void unbind(final Contract contract) {
        assert contract != null;

        Dataset dataset;
        String projectCode;

        projectCode = contract.getProject() != null ? contract.getProject().getCode() : null;

        Collection<Project> projects = this.repository.findlAllPublishedProjects();

        SelectChoices options;

        Project project = contract.getProject() != null ? contract.getProject() : (Project)
projects.toArray()[0];

        options = SelectChoices.from(projects, "code", project);

        dataset = super.unbind(contract, "code", "project", "providerName", "customerName",
"instantiationMoment", "budget", "goals", "draftMode");

        dataset.put("project", projectCode);
        dataset.put("projects", options);

        super.getResponse().addData(dataset);
    }
}
```

Como se puede ver hay varios fragmentos con una cobertura menor a la ideal, la razón de esta situación es la siguiente:

Fragmento	Justificación de cobertura
`assert contract != null;`	Esta estructura es heredada de los proyectos de ejemplo y supone una situación que no es posible controlar por nosotros por que pertenece al funcionamiento del framework.
`if (!super.getBuffer().getErrors().hasErrors("budget") && contract.getBudget() != null)`	No es posible pasar todos los estados, en este caso que un presupuesto sea nulo y el campo del atributo no tenga errores.
`super.state(!contract.getProject().isDraftMode(), "project", "client.contract.form.error.project");`	Este fragmento supone una validación realizada con el fin de evitar un posible hackeo en el campo de proyecto, el usuario común nunca va a encontrar posibilidad de llegar a este estado usando el sistema de forma correcta.

Funcionalidad:	Eliminación de contratos
Clase:	**Descripción del testeo realizado**
delete.safe	Para este caso se ha accedido como el rol indicado (cliente) y se ha realizado la petición para eliminar un contrato específico, esto se ha llevado a cabo con varios usuarios de rol cliente y eliminando varios contratos por cada uno.
delete.hack	En este caso se ha realizado la imposición de la id de un contrato en el formulario de otro, lo que permite probar que un usuario con el rol correcto no pueda eliminar contratos que no son suyos, o que no deberían poder borrarse por que han sido publicados. En todas las circunstancias probadas se lanza un error 500 “access is not authorized”.
bugs encontrados gracias al testeo	No se han encontrado bugs durante la realización de estos tests.

El código realizado y la cobertura de este se puede ver abajo:

```
@Service
public class ClientContractDeleteService extends AbstractService<Client, Contract> {

    // Internal state ---------------------------------------------------------

    @Autowired
    private ClientContractRepository repository;

    // AbstractService interface ----------------------------------------------
    @Override
    public void authorise() {
        int contractId;
        Contract contract;
        int clientId;
        boolean isValid;

        contractId = super.getRequest().getData("id", int.class);
        contract = this.repository.findContractById(contractId);
        clientId = super.getRequest().getPrincipal().getActiveRoleId();

        isValid = clientId == contract.getClient().getId() && contract.isDraftMode();

        super.getResponse().setAuthorised(isValid);
    }

    @Override
    public void load() {
        Contract contract;
        Integer id;

        id = super.getRequest().getData("id", int.class);
        contract = this.repository.findContractById(id);

        super.getBuffer().addData(contract);
    }
```

```
    @Override
    public void bind(final Contract contract) {
        assert contract != null;

        Integer managerId = super.getRequest().getPrincipal().getActiveRoleId();
        Client client = this.repository.findClientById(managerId);

        contract.setClient(client);
        super.bind(contract, "code", "project", "providerName", "customerName", "instantiationMoment",
"budget", "goals");
    }

    @Override
    public void validate(final Contract contract) {
        assert contract != null;

    }

    @Override
    public void perform(final Contract contract) {
        assert contract != null;

        this.repository.delete(contract);
    }

}
```

Como se puede ver hay un fragmento con una cobertura marcada como amarilla, lo cual noes recomendable, la razón detrás de ello es :

Fragmento	Justificación de cobertura
`assert contract != null;`	Esta estructura es heredada de los proyectos de ejemplo y supone una situación que no es posible controlar por nosotros por que pertenece al funcionamiento del framework.

Ahora pasaremos a ver las funcionalidades relacionadas a la segunda entidad, Progress log

1.2 testeo funcional sobre la entidad progress log

Funcionalidad:	Creación de registros de progreso
Clase:	**Descripción del testeo realizado**
create.safe	Para este caso se ha seguido la metodología propuesta, enviar un formulario vacío y tras esto probar los distintos casos negativos

	según límites y validaciones, todo ello atributo a atributo y uno por cada vez. Con esta parte probada, se envían varios formularios válidos probando los distintos valores permitidos teniendo en cuenta límites establecidos y caracteres especiales (intentos de hackeo, charsets distintos o caracteres especiales).
create.hack	Para este caso se ha enviado la url perteneciente a la creación sin haber iniciado sesión, tras esto, se ha iniciado sesión como cliente y se ha enviado una petición para crear una entidad asociada a un contrato de otro cliente, por último, se ha hecho otra petición en la el rol es el correcto y el contrato pertenece a este, pero no esta publicado, siendo imposible la creación en estas circunstancias lanzando siempre un error 500 “access is not authorized”.
bugs encontrados gracias al testeo	Debido a que se había realizado un análisis en profundidad de los distintos requisitos realizados antes de comenzar a realizar tests, no se han encontrado bugs durante la realización de estos.

Se puede ver el código y la cobertura alcanzada abajo:

```
@Service
public class ClientProgressLogCreateService extends AbstractService<Client, ProgressLog> {

    // Internal state ---------------------------------------------------------

    @Autowired
    private ClientProgressLogRepository repository;

    // AbstractService interface ----------------------------------------------


    @Override
    public void authorise() {
        int contractId;
        Contract contract;
        int clientId;
        boolean isValid;

        contractId = super.getRequest().getData("contractId", int.class);
        contract = this.repository.findContractById(contractId);
        clientId = super.getRequest().getPrincipal().getActiveRoleId();

        isValid = clientId == contract.getClient().getId() && !contract.isDraftMode();

        super.getResponse().setAuthorised(isValid);

    }

    @Override
    public void load() {
        ProgressLog progressLog;

        progressLog = new ProgressLog();
        Integer clientId = super.getRequest().getPrincipal().getActiveRoleId();
        Client client = this.repository.findClientById(clientId);

        int contractId = super.getRequest().getData("contractId", int.class);
        Contract contract = this.repository.findContractById(contractId);

        progressLog.setClient(client);
        progressLog.setContract(contract);
        progressLog.setDraftMode(true);

        super.getBuffer().addData(progressLog);
```

```
    }

    @Override
    public void bind(final ProgressLog progressLog) {
        assert progressLog != null;

        super.bind(progressLog, "recordId", "completeness", "comment", "registrationMoment",
"responsiblePerson");

    }

    @Override
    public void validate(final ProgressLog progressLog) {
        assert progressLog != null;

        if (!super.getBuffer().getErrors().hasErrors("recordId")) {

            ProgressLog progressLogWithCode =
this.repository.findProgressLogByRecordId(progressLog.getRecordId());

            super.state(progressLogWithCode == null, "recordId", "client.progress-log.form.error.recordId");
        }

        if (!super.getBuffer().getErrors().hasErrors("registrationMoment")) {
            Date contractDate = progressLog.getContract().getInstantiationMoment();
            Date plDate = progressLog.getRegistrationMoment();

            Boolean isAfter = plDate.after(contractDate);
            super.state(isAfter, "registrationMoment", "client.progress-log.form.error.registrationMoment");
        }

        if (!super.getBuffer().getErrors().hasErrors("contract")) {
            Integer contractId;
            Contract contract;

            contractId = super.getRequest().getData("contractId", int.class);
            contract = this.repository.findContractById(contractId);

            super.state(!contract.isDraftMode(), "*", "client.progress-log.form.error.unpublished-contract");
        }
    }

    @Override
    public void perform(final ProgressLog progressLog) {
        assert progressLog != null;
        progressLog.setId(0);
        this.repository.save(progressLog);
    }

    @Override
    public void unbind(final ProgressLog progressLog) {
        assert progressLog != null;

        Dataset dataset;

        dataset = super.unbind(progressLog, "recordId", "completeness", "comment", "registrationMoment",
"responsiblePerson");
        dataset.put("contractId", super.getRequest().getData("contractId", int.class));

        super.getResponse().addData(dataset);
    }

}
```

En este caso, también existen varios fragmentos de código con una cobertura calificada de color amarilla, lo que denota que no se ha probado como debería, la razón de que suceda esto es la siguiente:

Fragmento	Justificación de cobertura

`assert progressLog != null;`	Esta estructura es heredada de los proyectos de ejemplo y supone una situación que no es posible controlar por nosotros por que pertenece al funcionamiento del framework.
`if (!super.getBuffer().getErrors().hasErrors("contract"))`	En este fragmento se declara que el atributo contrato no debe tener errores si se quiere entrar dentro del bloque if, dado que el contrato es asignado automáticamente a la hora de crear un registro de progreso, no es posible probar el caso en el que este atributo contenga errores si no es mediante hacking directo a este campo (POST hacking).
`super.state(!contract.isDraftMode(), "*", "client.progress-log.form.error.unpublished-contract");`	Este fragmento supone una validación realizada con el fin de evitar un posible hackeo en el campo de contrato, el usuario común nunca va a encontrar posibilidad de llegar a este estado usando el sistema de forma correcta ya que el contrato es asociado automáticamente y no le permite crear un registro de progreso a menos que el contrato ya esté publicado.

Funcionalidad:	Listado de registros de progreso
Clase:	**Descripción del testeo realizado**
list.safe	En este caso se ha probado la funcionalidad iniciando sesión con un rol adecuado y solicitando el listado de registros asociados a un contrato publicado en concreto, esta acción ha sido realizada para los distintos usuarios que tenían contratos publicados para listar los registros asociados a cada uno.
list.hack	Para este caso se ha enviado la url para listar los registros asociados a un determinado contrato sin haber iniciado sesión, tras esto, se ha iniciado sesión como cliente y se ha enviado una petición para listar los registros asociados a un contrato de otro cliente (comprobando contratos publicados y en modo borrador), por último, se ha hecho otra petición en la que el rol es el correcto y el contrato pertenece a este, pero no esta publicado, lo que imposibilita mostrar ningún listado.

bugs encontrados gracias al testeo	Durante el desarrollo de uno de los tests se encontró que se había realizado una asignación errónea de un registro de progreso a un contrato, lo que llevó a un error de pánico por parte del sistema, se analizó donde se encontraba el registro de progreso mal asignado y se asignó al contrato. Tras esto se realizaron los tests asociados a esta funcionalidad otra vez y se comprobó que todo funcionaba como debía.

Se puede ver el código y la cobertura alcanzada abajo:

```
@Service
public class ClientProgressLogListService extends AbstractService<Client, ProgressLog> {

    // Internal state ---------------------------------------------------------

    @Autowired
    private ClientProgressLogRepository repository;

    // AbstractService interface ----------------------------------------------
    @Override
    public void authorise() {
        int contractId;
        Contract contract;
        int clientId;
        boolean isValid;

        contractId = super.getRequest().getData("contractId", int.class);
        contract = this.repository.findContractById(contractId);
        clientId = super.getRequest().getPrincipal().getActiveRoleId();

        isValid = clientId == contract.getClient().getId()&& !contract.isDraftMode();

        super.getResponse().setAuthorised(isValid);

    }

    @Override
    public void load() {

        Collection<ProgressLog> progressLogs;
        int contractId;

        contractId = super.getRequest().getData("contractId", int.class);
        progressLogs = this.repository.findProgressLogsByContractId(contractId);

        super.getBuffer().addData(progressLogs);
    }

    @Override
    public void unbind(final ProgressLog progressLog) {

        assert progressLog != null;
        Dataset dataset;

        dataset = super.unbind(progressLog, "recordId", "draftMode", "completeness", "comment",
"registrationMoment", "responsiblePerson");

        super.getResponse().addData(dataset);
    }

    @Override
    public void unbind(final Collection<ProgressLog> progressLogs) {
        assert progressLogs != null;

        int contractId;

        contractId = super.getRequest().getData("contractId", int.class);

        super.getResponse().addGlobal("contractId", contractId);
    }
```

```
}
```

Se puede apreciar que el fragmento cuyo nivel de cobertura es clasificado con el codigo de color amarillo es el mismo que se ha ido repitiendo a lo largo de todo el proyecto, la causa de esta situación puede leerse mas abajo:

Fragmento	Justificación de cobertura
`assert progressLog != null;`	Esta estructura es heredada de los proyectos de ejemplo y pertenece al funcionamiento del framework

Funcionalidad:	Mostrar detalles de registro de progreso
Clase:	**Descripción del testeo realizado**
show.safe	Para este caso se ha accedido como el rol indicado (cliente) y tras listar los registros de progreso asociados a un contrato específico se ha solicitado los detalles del registro, esto se ha llevado a cabo con varios usuarios de rol cliente y mostrando los detalles de varios registros por cada usuario y contrato publicado.
show.hack	Para este caso se ha enviado una url solicitando los detalles de un registro sin haber iniciado sesión, tras esto, se ha iniciado sesión con un rol distinto(administrator) y se ha vuelto a enviar la petición para comprobar que no deja acceder a esta, por último, se ha iniciado sesión con un usuario del rol correcto y se ha realizado una petición dirigida a un contrato que no le pertenece, mostrando siempre un error 500 “access is not authorized”.
bugs encontrados gracias al testeo	Debido a que se había realizado un análisis en profundidad de los distintos requisitos realizados antes de comenzar a realizar tests, no se han encontrado bugs durante la realización de estos.

Se puede ver el código desarrollado y la cobertura relacionada a este justo abajo:

```
@Service
public class ClientProgressLogShowService extends AbstractService<Client, ProgressLog> {

    // Internal state ---------------------------------------------------------

    @Autowired
    private ClientProgressLogRepository repository;

    // AbstractService interface ----------------------------------------------


    @Override
    public void authorise() {
        int progressLogId;
        ProgressLog progressLog;
        int clientId;
```

```
        boolean isValid;

        progressLogId = super.getRequest().getData("id", int.class);
        progressLog = this.repository.findProgressLogById(progressLogId);
        clientId = super.getRequest().getPrincipal().getActiveRoleId();

        isValid = clientId == progressLog.getClient().getId();

        super.getResponse().setAuthorised(isValid);
    }

    @Override
    public void load() {

        ProgressLog progressLog;
        int progressLogId;

        progressLogId = super.getRequest().getData("id", int.class);
        progressLog = this.repository.findProgressLogById(progressLogId);

        super.getBuffer().addData(progressLog);
    }

    @Override
    public void unbind(final ProgressLog progressLog) {

        assert progressLog != null;
        Dataset dataset;

        dataset = super.unbind(progressLog, "recordId", "draftMode", "completeness", "comment",
"registrationMoment", "responsiblePerson");

        super.getResponse().addData(dataset);
    }
}
```

los fragmentos que no han sido cubiertos completamente durante la grabación de los tests y la justificación de esta situación puede verse más abajo:

Fragmento	Justificación de cobertura
`assert progressLog != null;`	Esta estructura es heredada de los proyectos de ejemplo y pertenece al funcionamiento del framework

Funcionalidad:	Actualización de registros de progreso
Clase:	**Descripción del testeo realizado**
update.safe	Para este caso se ha seguido la metodología propuesta en la que se envía un formulario vacío, tras esto se han probando los distintos casos negativos atributo a atributo, teniendo en cuenta los límites y validaciones introducidas.

	Una vez probado todo, se han hecho varios formularios de actualización correctos que han sido enviados. Estos contenían datos que probaban los distintos rangos de valores permitidos para los atributos, teniendo en cuenta el uso de caracteres especiales, estructuras potencialmente maliciosas (intentos de hackeo en los campos de texto) o charsets diferentes.
update.hack	En este caso se ha realizado una modificación en los formularios enviados con el fin de alterar la id del registro de progreso que se estaba modificando, probando que un usuario con el rol correcto no pudiera alterar información de registros que no son suyos, o que no deberían poder modificarse (registros ya publicados), devolviendose en estas circunstancias un error 500 “access is not authorized”.
bugs encontrados gracias al testeo	Debido a que se había realizado un análisis en profundidad de los distintos requisitos antes de comenzar a realizar tests, no se han encontrado bugs durante la realización de estos.

Podemos ver el código realizado y la cobertura que se ha alcanzado durante los tests más abajo:

```
@Service
public class ClientProgressLogUpdateService extends AbstractService<Client, ProgressLog> {

    // Internal state ---------------------------------------------------------

    @Autowired
    private ClientProgressLogRepository repository;

    // AbstractService interface ----------------------------------------------


    @Override
    public void authorise() {
        int progressLogId;
        ProgressLog progressLog;
        int clientId;
        boolean isValid;

        progressLogId = super.getRequest().getData("id", int.class);
        progressLog = this.repository.findProgressLogById(progressLogId);
        clientId = super.getRequest().getPrincipal().getActiveRoleId();

        isValid = clientId == progressLog.getClient().getId() && progressLog.isDraftMode();

        super.getResponse().setAuthorised(isValid);
    }

    @Override
    public void load() {

        ProgressLog progressLog;
        int progressLogId;

        progressLogId = super.getRequest().getData("id", int.class);
        progressLog = this.repository.findProgressLogById(progressLogId);

        super.getBuffer().addData(progressLog);
    }

    @Override
    public void bind(final ProgressLog progressLog) {
        assert progressLog != null;
```

```
        super.bind(progressLog, "recordId", "completeness", "comment", "registrationMoment",
"responsiblePerson");
    }

    @Override
    public void validate(final ProgressLog progressLog) {
        assert progressLog != null;

        if (!super.getBuffer().getErrors().hasErrors("recordId")) {

            ProgressLog progressLogWithCode =
this.repository.findProgressLogByRecordId(progressLog.getRecordId());

            if (progressLogWithCode != null)
                super.state(progressLogWithCode.getId() == progressLog.getId(), "recordId",
"client.progress-log.form.error.recordId");
        }

        if (!super.getBuffer().getErrors().hasErrors("registrationMoment")) {

            Date contractDate = progressLog.getContract().getInstantiationMoment();
            Date plDate = progressLog.getRegistrationMoment();

            Boolean isAfter = plDate.after(contractDate);

            super.state(isAfter, "registrationMoment", "client.progress-log.form.error.registrationMoment");
        }

        if (!super.getBuffer().getErrors().hasErrors("contract")) {
            Contract contract;

            contract = progressLog.getContract();

            super.state(!contract.isDraftMode(), "*", "client.progress-log.form.error.unpublished-contract");
        }

    }

    @Override
    public void perform(final ProgressLog progressLog) {
        assert progressLog != null;

        this.repository.save(progressLog);
    }

    @Override
    public void unbind(final ProgressLog progressLog) {
        assert progressLog != null;

        Dataset dataset;

        dataset = super.unbind(progressLog, "recordId", "completeness", "comment", "registrationMoment",
"responsiblePerson", "draftMode");

        super.getResponse().addData(dataset);
    }
}
```

Si nos fijamos en el código realizado, hay varios fragmentos cuya cobertura no es perfecta, la razón y el fragmento en concreto se puede leer más abajo:

Fragmento	Justificación de cobertura
`assert progressLog != null;`	Esta estructura es heredada de los proyectos de ejemplo y supone una situación que no es posible controlar por nosotros por que pertenece al funcionamiento del framework.

`if (!super.getBuffer().getErrors().hasErrors("contract"))`	Este if declara que el el atributo contrato no debe tener errores, dado que el contrato es asignado automáticamente a la hora de crear un registro de progreso, no hay posibilidad de probar el caso en el que este atributo contenga errores si no es mediante hacking directo a este campo (POST hacking)
`super.state(!contract.isDraftMode(), "*", "client.progress-log.form.error.unpublished-contract");`	Este fragmento supone una validación realizada con el fin de evitar un posible hackeo en el campo de contrato, el usuario común nunca va a encontrar posibilidad de llegar a este estado usando el sistema de forma correcta ya que el contrato es asociado automáticamente y no le permite crear un registro de progreso a menos que el contrato ya esté publicado.

Funcionalidad:	Publicación de registros de progreso
Clase:	**Descripción del testeo realizado**
publish.safe	Para este caso se ha seguido la metodología propuesta en la que se envía un formulario vacío, y tras esto se van probando los distintos casos negativos atributo a atributo y uno por cada vez. Una vez probado todo, se han enviado varios formularios de publicación correctos probando los distintos valores permitidos teniendo en cuenta límites y caracteres especiales (intentos de hackeo, charsets distintos o caracteres especiales).
publish.hack	En este caso se ha realizado una modificación en los formularios enviados con el fin cambiar la id del contrato al que se hace referencia en el formulario, probando que un usuario con el rol correcto no pudiera publicar contratos que no son suyos, o que no deberían poder publicarse porque ya lo están. En todos los casos probados se devuelve un error 500 “access is not authorized”.

bugs encontrados gracias al testeo	Gracias a una revisión a conciencia de esta funcionalidad, no se han encontrado bugs durante el desarrollo de los tests, corroborando la efectividad de la revisión realizada.

Una vez visto la forma de realizar los tests para esta funcionalidad podemos ver el código realizado y la cobertura lograda con dichos tests abajo:

```
@Service
public class ClientProgressLogPublishService extends AbstractService<Client, ProgressLog> {

    // Internal state ---------------------------------------------------------
    @Autowired
    private ClientProgressLogRepository repository;

    // AbstractService interface ----------------------------------------------
    @Override
    public void authorise() {
        int progressLogId;
        ProgressLog progressLog;
        int clientId;
        boolean isValid;

        progressLogId = super.getRequest().getData("id", int.class);
        progressLog = this.repository.findProgressLogById(progressLogId);
        clientId = super.getRequest().getPrincipal().getActiveRoleId();

        isValid = clientId == progressLog.getClient().getId() && progressLog.isDraftMode();

        super.getResponse().setAuthorised(isValid);
    }

    @Override
    public void bind(final ProgressLog progressLog) {
        assert progressLog != null;

        super.bind(progressLog, "recordId", "completeness", "comment", "registrationMoment",
"responsiblePerson");
    }

    @Override
    public void validate(final ProgressLog progressLog) {
        assert progressLog != null;

        if (!super.getBuffer().getErrors().hasErrors("recordId")) {

            ProgressLog progressLogWithCode =
this.repository.findProgressLogByRecordId(progressLog.getRecordId());

            if (progressLogWithCode != null)
                super.state(progressLogWithCode.getId() == progressLog.getId(), "recordId",
"client.progress-log.form.error.recordId");

        }

        if (!super.getBuffer().getErrors().hasErrors("registrationMoment")) {

            Date contractDate = progressLog.getContract().getInstantiationMoment();
            Date plDate = progressLog.getRegistrationMoment();

            Boolean isAfter = plDate.after(contractDate);
            super.state(isAfter, "registrationMoment", "client.progress-log.form.error.registrationMoment");
        }

        if (!super.getBuffer().getErrors().hasErrors("contract")) {
            Contract contract;

            contract = progressLog.getContract();

            super.state(!contract.isDraftMode(), "*", "client.progress-log.form.error.unpublished-contract");
        }
```

```
        if (!super.getBuffer().getErrors().hasErrors("completeness")) {

            ProgressLog log =
this.repository.findContractProgressLogWithMaxCompleteness(progressLog.getContract().getId());

            if (log != null)
                super.state(log.getCompleteness() < progressLog.getCompleteness(), "completeness",
"client.progress-log.form.error.completeness");

        }

        if (!super.getBuffer().getErrors().hasErrors("registrationMoment")) {

            ProgressLog log =
this.repository.findContractProgressLogWithMaxCompleteness(progressLog.getContract().getId());

            if (log != null)
                super.state(log.getRegistrationMoment().before(progressLog.getRegistrationMoment()),
"registrationMoment", "client.progress-log.form.error.sameMoment");
        }
    }

    @Override
    public void load() {

        ProgressLog progressLog;
        int progressLogId;

        progressLogId = super.getRequest().getData("id", int.class);
        progressLog = this.repository.findProgressLogById(progressLogId);

        super.getBuffer().addData(progressLog);
    }

    @Override
    public void perform(final ProgressLog progressLog) {
        assert progressLog != null;

        progressLog.setDraftMode(false);
        this.repository.save(progressLog);
    }

    @Override
    public void unbind(final ProgressLog progressLog) {
        assert progressLog != null;
        Dataset dataset;

        dataset = super.unbind(progressLog, "recordId", "completeness", "comment", "registrationMoment",
"responsiblePerson", "draftMode");

        super.getResponse().addData(dataset);
    }
}
```

Para este caso también existen fragmentos de código cuya cobertura no es perfecta, las razones y los fragmentos concretos son los siguientes:

Fragmento	Justificación de cobertura
`assert progressLog != null;`	Esta estructura es heredada de los proyectos de ejemplo y supone una situación que no es posible controlar por nosotros por que pertenece al funcionamiento del framework.

`if (!super.getBuffer().getErrors().hasErrors("contract"))`	Este if declara que el el atributo contrato no debe tener errores, dado que el contrato es asignado automáticamente a la hora de crear un registro de progreso, no hay posibilidad de probar el caso en el que este atributo contenga errores si no es mediante hacking directo a este campo (POST hacking).
`super.state(!contract.isDraftMode(), "*", "client.progress-log.form.error.unpublished-contract");`	Este fragmento supone una validación realizada con el fin de evitar un posible hackeo en el campo de contrato, el usuario común nunca va a encontrar posibilidad de llegar a este estado usando el sistema de forma correcta ya que el contrato es asociado automáticamente y no le permite crear un registro de progreso a menos que el contrato ya esté publicado.

Funcionalidad:	Eliminación de registros de progreso
Clase:	**Descripción del testeo realizado**
delete.safe	Para este caso se ha accedido como el rol indicado (cliente) y se ha realizado la petición para eliminar un registro de progreso específico, se han borrado varios registros de progreso pertenecientes a diferentes clientes y contratos de estos.
delete.hack	En este caso se ha aprovechado el formulario de actualización de un registro de progreso para, cambiando la id del registro guardada en el formulario, intentar eliminar el registro con dicha id. Se ha probado que un usuario con el rol correcto no pueda eliminar contratos que no son suyos, o que no deberían borrarse por que han sido publicados, devolviendo siempre un error 500 “access is not authorized” como respuesta a este tipo de peticiones.
bugs encontrados gracias al testeo	No se han encontrado bugs durante la realización de estos tests.

El código realizado y la cobertura de este se puede ver abajo:

```
@Service
public class ClientProgressLogDeleteService extends AbstractService<Client, ProgressLog> {

    // Internal state ---------------------------------------------------------

    @Autowired
    private ClientProgressLogRepository repository;
    // AbstractService interface ----------------------------------------------

    @Override
    public void authorise() {
```

```
        int progressLogId;
        ProgressLog progressLog;
        int clientId;
        boolean isValid;

        progressLogId = super.getRequest().getData("id", int.class);
        progressLog = this.repository.findProgressLogById(progressLogId);
        clientId = super.getRequest().getPrincipal().getActiveRoleId();

        isValid = clientId == progressLog.getClient().getId() && progressLog.isDraftMode();

        super.getResponse().setAuthorised(isValid);
    }

    @Override
    public void load() {

        ProgressLog progressLog;
        int progressLogId;

        progressLogId = super.getRequest().getData("id", int.class);
        progressLog = this.repository.findProgressLogById(progressLogId);

        super.getBuffer().addData(progressLog);
    }

    @Override
    public void bind(final ProgressLog progressLog) {
        assert progressLog != null;

        Integer clientId = super.getRequest().getPrincipal().getActiveRoleId();
        Client client = this.repository.findClientById(clientId);

        progressLog.setClient(client);
        super.bind(progressLog, "recordId", "completeness", "comment", "registrationMoment",
"responsiblePerson");
    }

    @Override
    public void validate(final ProgressLog progressLog) {
        assert progressLog != null;

    }

    @Override
    public void perform(final ProgressLog progressLog) {
        assert progressLog != null;

        this.repository.delete(progressLog);
    }

}
```

Las causas de que existan algunos fragmentos de código con una cobertura deficiente pueden leerse abajo:

Fragmento	Justificación de cobertura
`assert progressLog != null;`	Esta estructura es heredada de los proyectos de ejemplo y supone una situación que no es posible controlar por nosotros por que pertenece al funcionamiento del framework.

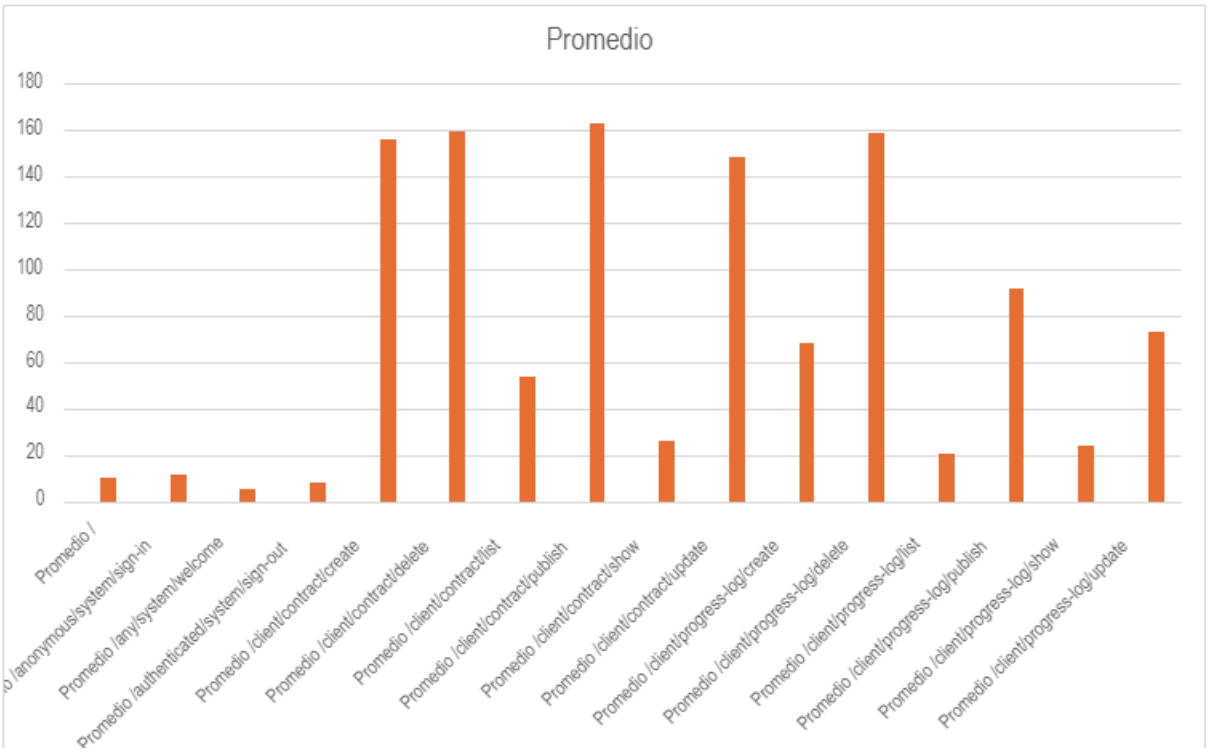
2. ANÁLISIS DEL RENDIMIENTO DE LOS TESTS

En esta sección del documento se procederá a dar las métricas que se han sacado del rendimiento del proyecto creado y se analizarán los resultados obtenidos con los cambios realizados para mejorar el rendimiento y con el equipo de otro compañero. Tras la aportación de las métricas se darán las hipótesis de los resultados obtenidos y las conclusiones que podemos sacar de estos.

Las métricas pertenecientes al análisis del rendimiento del sistema durante los tests realizados han sido las siguientes:

Estadísticas				
Media	51,3699597	Intervalo(ms)	47,3157524	55,424167
Error típico	2,06650823	Intervalo(s)	0,04731575	0,05542417
Mediana	20,0168			
Moda	4,8247			
Desviación es	73,0913181			
Varianza de la	5342,34078			
Curtosis	20,305202			
Coeficiente d	3,41077637			
Rango	839,9084			
Mínimo	1,8698			
Máximo	841,7782			
Suma	64263,8196			
Cuenta	1251			
Nivel de conf	4,05420729			

En términos generales, los resultados muestran unos tiempos bastante altos, lo que podría denotar problemas de rendimiento en el equipo o un código poco eficiente, si miramos el gráfico del tiempo medio de las distintas funcionalidades:



Podremos apreciar que las peticiones más ineficientes pertenecen a las funcionalidades de creación, borrado y publicación de contratos, mientras que para la entidad secundaria, progress logs estas funcionalidades son el borrado y actualización de estos.

2.1 Análisis del rendimiento del sistema sin índices frente a la implementación de estos

Como se explicó antes, debido a los resultados del rendimiento del sistema se ha estudiado la introducción de índices con el fin de mejorar estos tiempos durante el desarrollo de las pruebas, una vez introducidos, se repitieron los cálculos y los resultados obtenidos frente a los anteriores quedarían así:

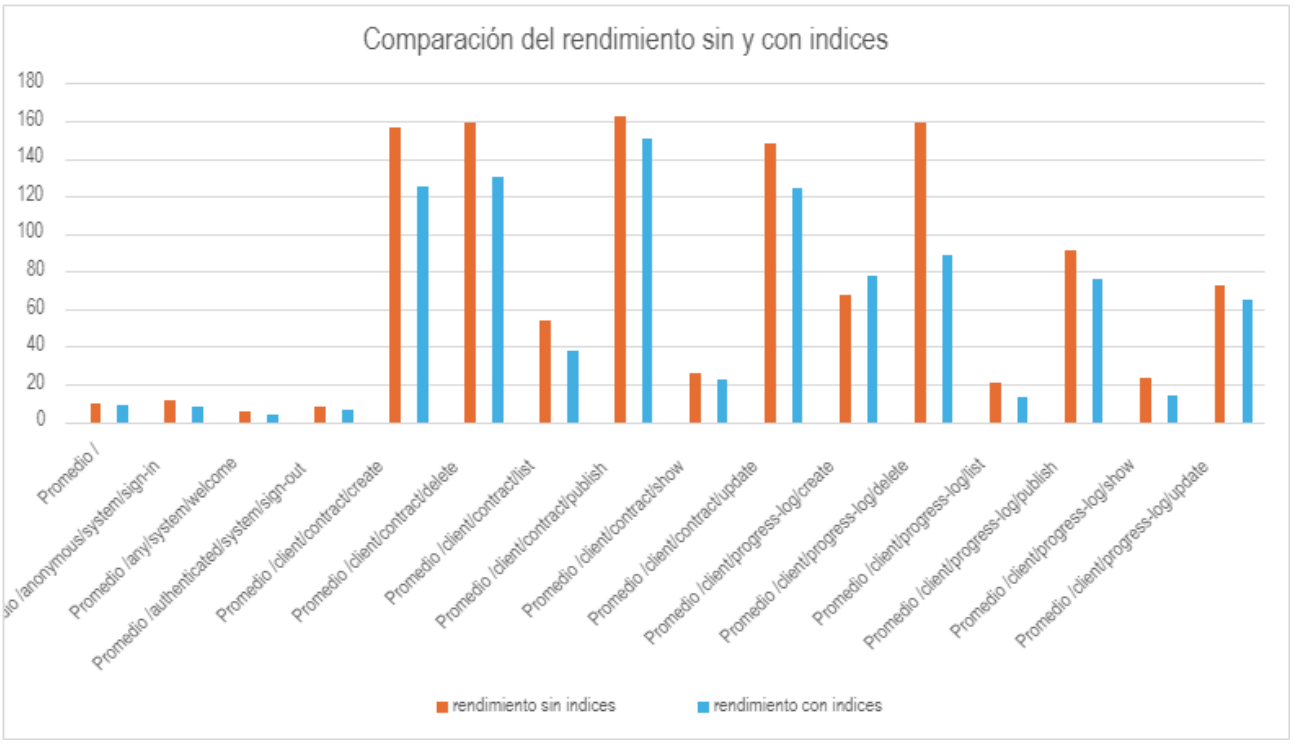
Antes			*Después*		
Media	51,6502642		Media	41,1110389	
Error típico	2,04915882		Error típico	1,94842244	
Mediana	20,25045		Mediana	14,5537	
Moda	4,8247		Moda	3,7217	
Desviación es	72,9684712		Desviación es	68,9146855	
Varianza de la	5324,39779		Varianza de la	4749,23388	
Curtosis	20,1123686		Curtosis	148,292076	
Coeficiente d	3,38195269		Coeficiente d	8,62061546	
Rango	839,9084		Rango	1444,6426	
Mínimo	1,8698		Mínimo	1,857	
Máximo	841,7782		Máximo	1446,4996	
Suma	65492,535		Suma	51429,9097	
Cuenta	1268		Cuenta	1251	
Nivel de confi	4,02011785		Nivel de confi	3,82253909	
Intervalo(ms)	47,6301463	55,670382	Intervalo(ms)	37,2884998	44,933578
Intervalo(s)	0,04763015	0,05567038	Intervalo(s)	0,0372885	0,04493358

Como se puede ver, la introducción de los índices ha mejorado los datos que estas tablas ofrecen respecto a los valores anteriores, y si nos fijamos en las métricas aportadas por los cálculos del Z-test para el intervalo de confianza que se estableció (95%)

	Antes	Después de indices
	182,3963	*345,0539*
Media	51,54707079	40,86788464
Varianza (conocida)	5324,39779	4749,23388
Observaciones	1267	1250
Diferencia hipotética de las	0	
z	3,775248832	
P(Z<=z) una cola	7,9924E-05	
Valor crítico de z (una cola)	1,644853627	
Valor crítico de z (dos colas	0,000159848	
Valor crítico de z (dos colas	1,959963985	

Podremos ver que la métrica del valor crítico de z (dos colas) está entre 0 y el valor de alfa (0.05 en nuestro caso) lo que nos permite decir con seguridad que se pueden comparar las medias del rendimiento entre la versión con índices y sin ellos.

Para la comparación se ha realizado una tabla que se puede consultar más abajo:



Como se puede apreciar, se produce una mejora notable en los tiempos medios para los distintos métodos, viéndose la mayor diferencia en en el borrado de la entidad secundaria progress log, cuyo tiempo medio se ve reducido casi a la mitad gracias a la introducción de los índices. Sin embargo, a pesar de que los índices han mejorado en general los tiempos respecto a su ausencia, en el caso específico de la creación de registros de progreso se ha visto aumentado.

Atendiendo a todos estos datos que se han aportado, nos es posible afirmar que la introducción de los índices en el proyecto ha traído más beneficios que inconvenientes, resultando ser una medida bastante fácil de implementar considerando la mejora que ha producido. Sin embargo, si continuamos leyendo los análisis realizados, se puede apreciar que el equipo con el que he realizado el proyecto sigue teniendo un rendimiento bastante peor.

2.2 Análisis y comparación del rendimiento entre 2 sistemas

Ahora veremos los análisis realizados comparando el rendimiento de 2 equipos, el equipo donde se ha realizado la entrega (el estudiante 2) y el ordenador de otro miembro del equipo, el estudiante 5:

PC_STUDENT02	
Media	51,6502642
Error típico	2,04915882
Mediana	20,25045
Moda	4,8247
Desviación es	72,9684712
Varianza de la	5324,39779
Curtosis	20,1123686
Coeficiente d	3,38195269
Rango	839,9084
Mínimo	1,8698
Máximo	841,7782
Suma	65492,535
Cuenta	1268
Nivel de confi	4,02011785

Intervalo(ms)	47,6301463	55,670382
Intervalo(s)	0,04763015	0,05567038

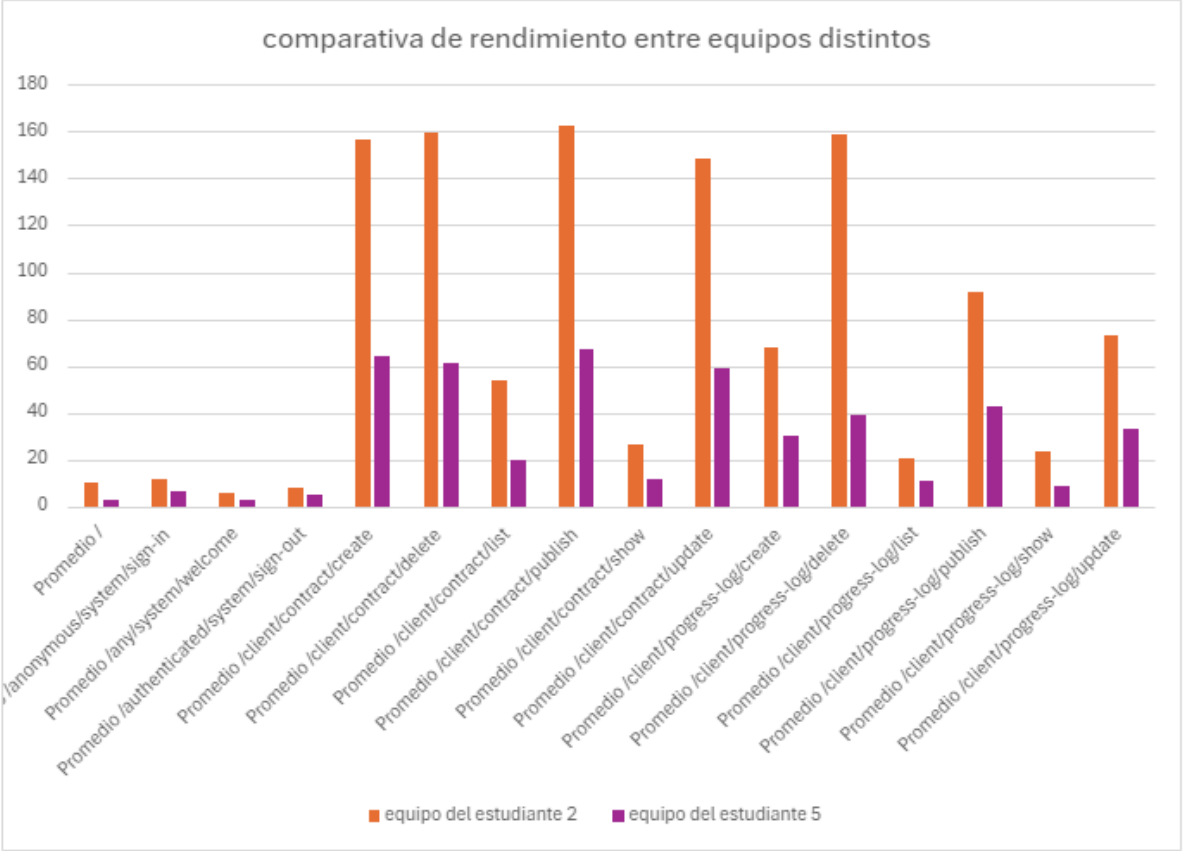
PC_STUDENT05	
Media	21,8061835
Error típico	0,65130727
Mediana	11,137399
Moda	2,436199
Desviación es	23,0363983
Varianza de la	530,675645
Curtosis	5,06988594
Coeficiente d	1,86185733
Rango	210,331001
Mínimo	1,4961
Máximo	211,827101
Suma	27279,5356
Cuenta	1251
Nivel de confi	1,27777602

Intervalo(ms)	20,5284075	23,0839596
Intervalo(s)	0,02052841	0,02308396

Si nos fijamos en ambas columnas, se podrá ver que existe una gran diferencia entre ambos equipos, viéndose una reducción de la media de más del 50%.
Para seguir analizando y dar un veredicto de los análisis, primero debemos comprobar que los cálculos del Z-test sean correctos

	PC_STUDENT02	PC_STUDENT05
	182,3963	*118,9223*
Media	51,54707079	21,72849065
Varianza (conocida)	5324,39779	530,675645
Observaciones	1267	1250
Diferencia hipotética de las medias	0	
z	13,86250114	
P(Z<=z) una cola	0	
Valor crítico de z (una cola)	1,644853627	
Valor crítico de z (dos colas)	0	
Valor crítico de z (dos colas)	1,959963985	

Viendo la tabla que hemos sacado y fijándonos en el valor crítico de z (de 2 colas) calculado para nuestro intervalo de confianza, podemos asegurar que se encuentra entre 0 y alfa (0.05).
Este resultado nos permite asegurar que las comparaciones que se realicen entre las medias de los rendimiento de un equipo y otro serán válidas, por ello se ha realizado una tabla comparativa en la que se puede ver el rendimiento del sistema en el equipo del estudiante 2 frente al estudiante 5:



Viendo el contenido de la tabla, la diferencia es más que notable, ganando claramente el equipo del estudiante 5, en general se ve una reducción de los tiempos medios comprendida entre el 50-60% siendo uno de los casos más notables el de los tiempos para borrar un registro de progreso, donde la media para el equipo del estudiante 2 está cerca de los 160 ms, mientras que para el equipo del estudiante 5 no llega a los 40 ms.

3. Conclusiones

Tras realizar los análisis de la cobertura podemos sacar en claro que se ha cubierto la mayor parte del código realizado en las distintas entregas, existiendo algunas excepciones que han sido explicadas y justificadas, por otra parte, la realización de estos test nos ha permitido tener una serie de pruebas que se han usado para medir el rendimiento del equipo.

Gracias a los análisis realizados al proyecto mediante las herramientas que ofrece el framework y eclipse se ha podido dar una estimación del rendimiento del sistema, cuyos resultados han sido más altos de lo que creíamos, en base a ello, se han implementado los índices y se ha podido comprobar como la integración de estos en el código ha mejorado los tiempos medios.

Sin embargo, tras la realización de las comparaciones entre los tiempos de mi equipo con el de otro integrante pude comprobar que los tiempos que mi equipo ofrecía eran demasiado

altos en comparación, lo que me ha llevado a pensar que lo que originalmente creía que era un problema de optimización del código realizado, resultó ser un problema del ordenador usado, esta afirmación es bastante fácil de corroborar si atendemos a las especificaciones de los equipos involucrados, siendo mi equipo (estudiante 2) un ordenador con 7 años de antigüedad que se compró con el fin de realizar actividades de ofimática básica, mientras que el ordenador del estudiante 5 no solo es más actual, sino las prestaciones que ofrecen son bastante mejores comparadas a las que ofrecía mi equipo en el momento de la compra.

Gracias a estos análisis he podido entender cuáles eran los principales problemas de rendimiento del sistema desarrollado y se intentará poner solución ahorrando para comprar un equipo más actualizado que pueda usarse para desarrollos como este.