Solms Training, Consulting and Development

Physical: 113 Barry Hertzog Ave, Emmarentia, Johannesburg, South Africa
Postal: PostNet Suite no 237, Private Bax X9, Melville 2109, South Africa

Phone: +27 (11) 646-6459
Fax: +27 (11) 646-5868
World Wide Web: www.solms.co.za

URDAD for System Design

Use-Case Responsibility Driven Analysis and Design

Dr. Fritz Solms

Table of Contents

1. Introduction 1
  1.1. Design versus development processes 2
  1.2. Background 2
  1.3. Design versus Architecture 2
  1.4. Requirements for good design 2
    1.4.1. Benefits of adhering to these design principles 3
  1.5. URDAD drivers 4
2. URDAD: The Process 4
  2.1. Overview of URDAD 4
  2.2. Responsibility Identification 6
  2.3. Responsibility Allocation 7
  2.4. Specifying the collaboration 7
  2.5. Projecting out the context of the collaboration 9
  2.6. Service provider contracts 10
    2.6.1. Pre-Conditions 10
    2.6.2. Post-Conditions 10
    2.6.3. Invariance constraints 10
    2.6.4. Quality requirements 11
    2.6.5. Contracts and testing 11
    2.6.6. Contracts and the Object Constraint Language 11
    2.6.7. MailSender contract 11
  2.7. Value objects 11
  2.8. Transition to the next level of granularity 12
3. Summary and conclusions 13
Bibliography 14

Abstract

URDAD, Use-Case, Responsibility Driven Analysis and Design, provides a simple, intuitive design process which can be embedded within iterative, use-case driven software development methodologies like extreme programming or the Rational Unified Process (RUP). URDAD has been formulated in a way which encourages sound design principles. The process directly generates clean layers of granularity, with good responsibility localization across loosely components of minimal structural complexity. The resultant design is architecture and technology neutral. Following OMG's Model Driven Architecture (MDA) this platform independent design is mapped onto some choice of architecture and realization techonologies.

1. Introduction

Good design makes systems more flexible and robust, reduces maintenance cost and increases life

expectancy. Yet, generating a "*good design*" is a non-trivial task. This article looks at core attributes of good design and explains how URDAD provides a simple algorithmic design process which leads to a design satisfying many of the requirements of a good design.

1.1. Design versus development processes

Software developers are phased with the problem of having to provide a design and ultimately an implementation which realizes the use case requirements. Software development processes like RUP (the Rational Unified Process) and Extreme Programming provide a higher level process for effectively managing the risk and quality of the development process. They do not, however, provide a design process. One still has to decide how to design and integrate the system components in order to obtain the desired functionality.

URDAD addresses this problem by providing a process for taking a use case through to realization. This design process can be plugged into higher level software development processes.

1.2. Background

URDAD has grown out of Responsibility Driven Design (RDD) methodology pioneered by Rebecca Wirfs-Brock and Brian Wilkerson (see [Wirfs-Brock-Wilkerson-1989], [Wirfs-Brock-Wilkerson-Wiener-1990] and [Wirfs-Brock-McKean-2002]). Like RDD, URDAD focuses during the early stages of the design on identifying and assigning responsibilities. Also, like RDD, URDAD puts a lot of emphasis on client-server contracts. However, unlike RDD, URDAD critically requires that responsibilities should be identified before one identifies the objects which will ultimately host the responsibilities. Furthermore, RDD does not provide a framework which generates the different layers of granularity naturally and cleanly. The ability to look at use case realization at different levels of granularity is a major benefit of URDAD. Finally, URDAD provides a step-for-step algorithm for designing a system across its levels of granularity.

Other methods like the *ICONIX* process from Doug Rosenberg discussed in [Rosenberg-Scott-1999] provide a structured process for evolving the static model from the collaboration requirements, but are not really responsibility driven, nor do they project out clean layers of granularity.

1.3. Design versus Architecture

URDAD is a design process. It does not address architecture. Modern approaches view architecture as orthogonal to design. While design realizes the functional (use case) requirements, architecture addresses non-functional requirements like scaleability, reliability, security, modifiability and so on. These are called the quality attributes of the architecture. It is the architecture which ensures that these qualities are realized across the various use cases of the system. Architecture will, for example, specify whether clustering should be used to achieve availability, whether reliability should be guaranteed with session replication, whether thread, component and resource connection pooling should be used to improve performance and reduce resource demand, the choice of integration technologies and so on.

Technologies change quite frequently and a change in implementation technology should not require a change in design. URDAD generates a design which is architecture and technology neutral. This approach is aligned with OMG's *Model Driven Architecture* (MDA) -- see [Frankel-2003]. MDA requires that the design phase yields a *Platform Independent Model* (PIM) which ensures that design survives technologies and architectures. MDA then maps the PIM onto the chosen architecture and technologies resulting in the *Platform Specific Model* (PSM).

URDAD thus generates MDA's PIM. The PIM is then mapped onto the chosen architecture and technologies. This may be done manually or using MDA tools.

1.4. Requirements for good design

In order to be able to demonstrate that URDAD is a process which tends to lead to "*good design*", we first have to understand the core qualities of good design. Most of these are accepted design principles:

- **Responsibility localization.** A design with good responsibility localization is often referred to as a design with a high level of cohesion. In such a design each component adheres to the *single responsibility principle*; i.e. each component thus has only a single responsibility at some level of granularity and all its attributes and services are narrowly aligned with its responsibility.

- **Clean layers of granularity.** This very important aspect of good design enables one to work effectively at various levels of granularity. The layers should adhere to the dependency inversion principle, i.e. components in a lower level of granularity should not have any dependency on higher-level components. Also, one should be able to understand a higher-level workflow without having to understand the finer details. At any level of granularity the responsibilities should be well defined and the workflow should be self-contained and comprehensible.

  Note

  To illustrate the benefit of being able to understand a system at different levels of granularity, let us have a look at a car. A non-technical person can learn to effectively drive a car. This is only possible because they do not have to understand the details of the lower level functioning of the car. A mechanic in the local service station will have to understand the car at a lower level of granularity, but does not need to understand the finer details of how the gearbox works. Again, it would be difficult (and expensive) to source a mechanic who would be able to understand the functioning of the car at all levels of granularity. Finally a gearbox specialist will understand the system at an even lower level of granularity, without having to understand the higher-level workflows. The various levels of granularity facilitate that different role players can all function effectively without anybody having to understand the entire system.

- **Decoupling.** Decoupling provides a high level of flexibility and improves maintainability. If one component uses another component we effectively have a client-server relationship. Generally clients would not want to lock into a particular service provider. Instead, the client defines the requirements in a contract (in business modeling this would be an SLA). The contract specifies the services which service providers need to provide (the interface), the pre- and post-conditions for those services and the non-functional requirements.

- **Simplicity.** If everything else is equal, then the simpler solution is preferable. Complexity results in increased development costs, risk and maintenance costs. A design which is understandable and conceptually intuitive is preferable above one which is difficult to explain and non-intuitive.

- **Architecture and technology neutral.** The design should remain valuable over a long period. To this end the design should be able to survive technologies, changes in access mechanisms and architectural changes. This is usually achieved by following the guidelines of OMG's *Model Driven Architecture*, (MDA), which suggests that the core design should be technology and architecture neutral and that this core design should then be mapped onto one's choice of technologies and architecture.

1.4.1. Benefits of adhering to these design principles

Adhering to the above design principles provides a range of short and long-term benefits to organizations including

- **Understandability.** Understandability is promoted by simplicity, good responsibility localization, intuitive naming, and the ability to view workflows at various levels of granularity.

- **Reusability.** Reusability is really a direct consequence of good responsibility localization together with a component based approach where components realize well defined contracts. Classes which address a particular combination of responsibilities relevant for a particular problem are not generally re-usable. On the other hand, classes whose services address only a single domain of responsibility and whose behaviour is well defined in a contract are generally much more likely to be re-usable.

- **Testability.** This is facilitated through specifying a contract for each component at any level of granularity.
- **Maintainability.** Simplicity, responsibility localization which results in localized maintenance, the ability to effectively work at different levels of granularity, decoupling, testability and re-usability all contribute to making a system maintainable.
- **Longevity.** A design which is architecture and technology neutral can survive changes in technologies and architecture. Furthermore, all the design principles which assist maintainability contribute also significantly to the longevity of the design.

1.5. URDAD drivers

URDAD aims to provide a simple design process which leads to good design. To this end it uses the requirements for a good design as direct drivers for the process.

- **Use-Case driven.** Virtually all modern software development processes are iterative, use case driven processes. They deliver value incrementally to users and clients through iterative realization of use cases. Furthermore, use case driven approaches deliver iteratively testable deliverables. Suitable design methodologies must hence similarly be use case driven, realizing the functional requirements provided with the use case requirements.
- **Good responsibility localization by design.** At each level of granularity URDAD starts by identifying the responsibilities at that level of granularity before identifying objects. Each responsibility is then assigned to a separate object. This approach yields good responsibility localization by design.
- **Simplicity: minimized structural complexity.** In URDAD one does not start with the design with the static model (the class diagrams). Instead, after having identified (via the responsibilities) the core components which collaborate to realize the use case at that level of granularity, one first looks at how they collaborate (the dynamics). The static model required to support the dynamics realizing the use case is then projected out from the dynamic model. The only structural features thus generated are those required to realize the use case.
- **Clean layers of granularity.** In URDAD a level of granularity is fixed by the responsibilities. The only components for a particular level of granularity will be those to which the core responsibilities were assigned. URDAD projects out the workflow as well as the static structure at a fixed level of granularity. Finally, URDAD provides a simple mechanism for stepping from the current to the next lower level of granularity.
- **Decoupling via contracts at all levels of granularity.** URDAD requires responsibility identification followed by responsibility allocation to core system components and core external service providers (actors). For each responsibility, irrespective of whether the responsibility is by an internal component or an actor, URDAD requires a contract. Contract are specified along the guidelines provided by *Design by Contract*.

In URDAD re-usability is viewed as a consequence of responsibility localization and contract (interface) based decoupling.

2. URDAD: The Process

URDAD assumes that one is following an iterative software development process where one realizes use cases iteratively. We thus assume that one has selected the use case(s) for the current iteration and that the use case requirements are available.

2.1. Overview of URDAD

URDAD generates MDA's *Platform Independent Model* (PIM) which is then mapped onto the chosen architecture and technologies to yield the *Platform Specific Model* (PSM). It takes use-case

based functional requirements through an iterative design process generating the various levels of granularity of the system iteratively.

Figure 1. Use-Case/Responsibility Driven Design

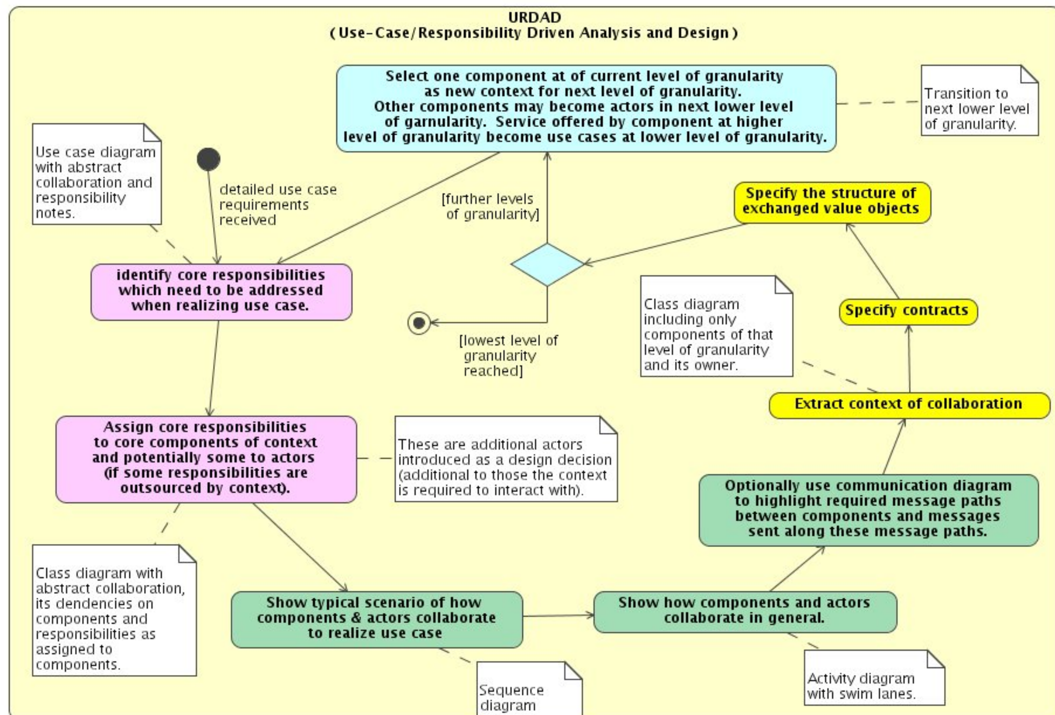


Figure 1, “Use-Case/Responsibility Driven Design” provides an overview of URDAD. The core steps of an iteration in URDAD are

1. Identify the core responsibilities which need to be addressed when realizing the use case.
2. Allocate each responsibility to either a component of the current context or an actor.
3. Specify how these components and actors collaborate to realize the use case.
4. Project out the context of the collaboration. This is that subset of the static model which at the current level of granularity is required to realize the use case.
5. Specify for each responsibility the contract they have to realize in the context of the current use case.
6. Specify the structure of exchanged value objects using class diagrams.
7. Traverse to the next lower level of granularity by selecting one of the components from the previous iteration as the new context with the services at the previous level of granularity becoming the use cases of this new, lower level of granularity.
8. Repeat the above steps for the use cases at the next lower level of granularity.

Note

URDAD is a double-iterative process with

1. *use case iterations* ensuring the system is designed iteratively, realizing use case after

use case, and

2. *design iterations* taking the design iteratively through lower and lower levels of granularity.

In the following sections these steps will be explained in detail, using the design of a simple mail client as an example.

2.2. Responsibility Identification

The first step of URDAD focuses on identifying the responsibilities which need to be addressed when realizing the use case. It is in many respects the most critical and most difficult step. This step fixes the level of granularity for the current design iteration. To this end it is important that the responsibilities identified are at the same level of granularity (or the same level of abstraction).

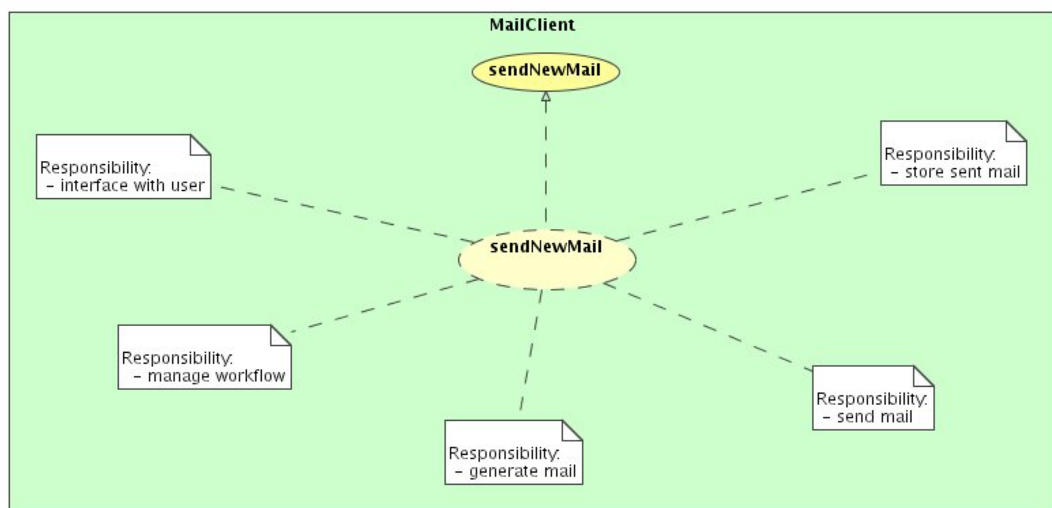
Note

URDAD requires that the *responsibilities should be identified before identifying system components*. As a second step these responsibilities will be assigned to core system components, ensuring good responsibility localization across system components.

Consider, as an example, a simple mail client. Assume we want to realize the *send-new-mail* use case. This use case will be realized through the collaboration of certain system components and potentially some actors. In Figure 2, "Responsibility identification." we identify the responsibilities which need to be addressed when realizing the use case. They include

- the workflow control and user interfacing responsibilities for the current context (the mail client as a whole),
- and the functional responsibilities for the mail client including that of generating the e-mail, sending it and storing the sent mail.

Figure 2. Responsibility identification.



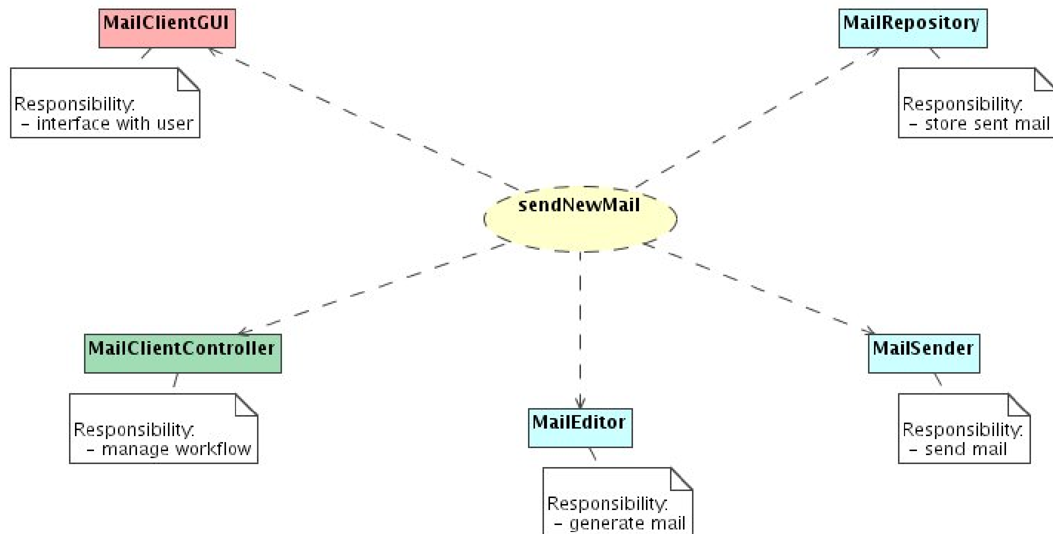
Responsibilities like that of managing addresses or marshaling the message on to the SMTP protocol are not relevant at this highest level of granularity. The responsibility of selecting addresses for an e-mail will be addressed in the context of creating the mail object and that of marshaling the mail onto the SMTP protocol will be addressed in the context of sending the e-mail. Hence both these re-

sponsibilities are lower level responsibilities which will be addressed at a lower level of granularity.

2.3. Responsibility Allocation

The second step of URDAD requires that each responsibility is assigned either to a separate system component or to an actor. URDAD thus enforces single responsibility principle by design. The objects thus introduced will all be at the same level of granularity as fixed by the responsibilities identified in the first step.

Figure 3. Responsibility allocation.



Revisiting our mail client we could potentially assign the responsibilities as illustrated in Figure 3, “Responsibility allocation.”.

2.4. Specifying the collaboration

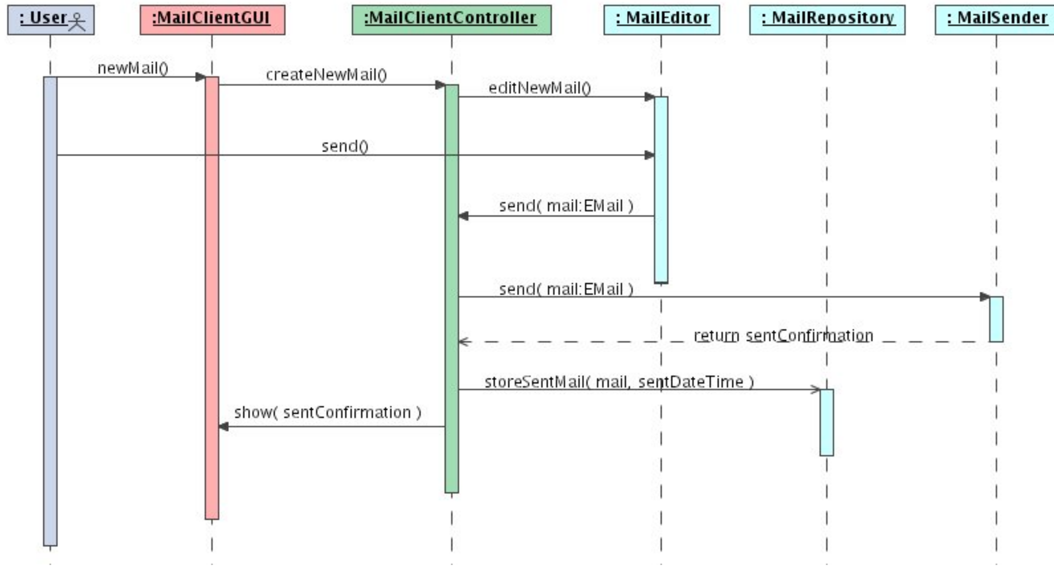
Having

1. identified the responsibilities which need to be addressed when realizing the use case, and
2. assigned these responsibilities to core system components and external service providers (actors),

we now need to look at how these components and actors collaborate to realize the use case.

Usually one first looks at a particular example (scenario) of realizing the use case. To this end one generally starts with a sequence diagram. Figure 4, “A scenario of realizing a use case at a specific level of granularity.” shows an example of such a sequence diagram.

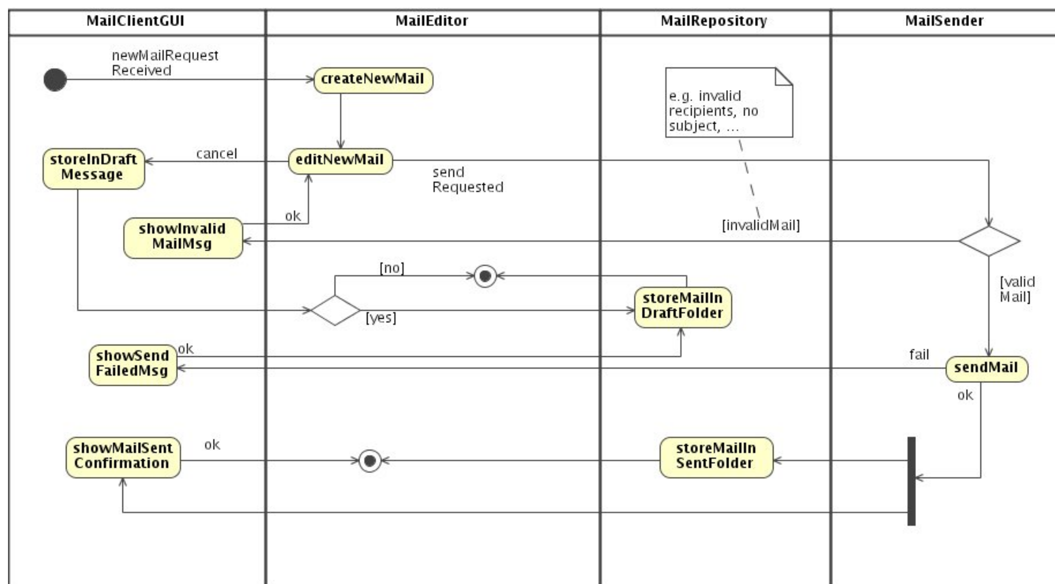
Figure 4. A scenario of realizing a use case at a specific level of granularity.



Note that the only objects participating in the collaboration shown in the sequence diagram are those which address the core responsibilities as identified for this level of granularity.

Once one is comfortable with a particular scenario (often a typical success scenario is chosen), one can look at the collaboration in general. This is commonly documented using a UML activity diagram. In Figure 5, “The use case collaboration in general.” we show an activity diagram documenting the general *send new mail* collaboration at the current level of granularity as fixed by the initial responsibility identification step.

Figure 5. The use case collaboration in general.

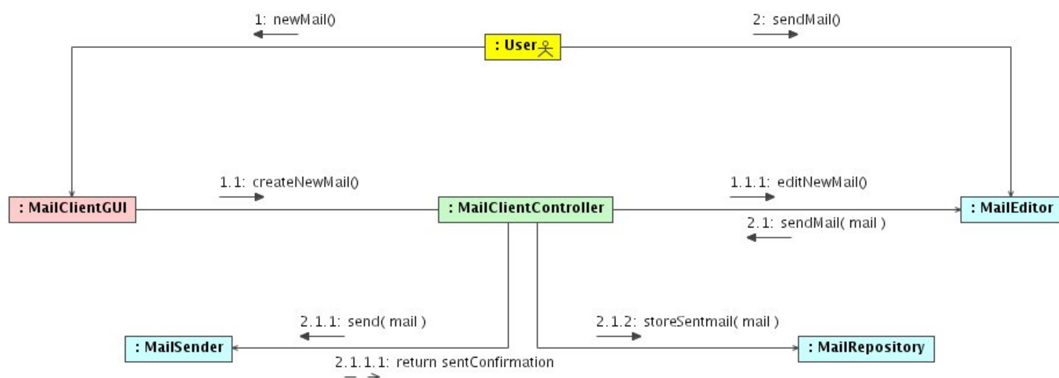


Finally, to simplify the transition to the collaboration context, one can use a UML communication diagram. It highlights the required communication paths as well as the service request messages sent along these paths and may contain other information from the static model. URDAD refrains from adding further structural information at this stage. The communication diagram corresponding to the sequence diagram in Figure 4, “A scenario of realizing a use case at a specific level of granularity.” is shown in Figure 6, “Communication diagram simplifying transition to collaboration context.”.

Note

Note that since URDAD only feeds message path information into communication diagram, the latter contains essentially the same information as the sequence diagram and may be auto-generated from the latter. This step may be (and often is) omitted.

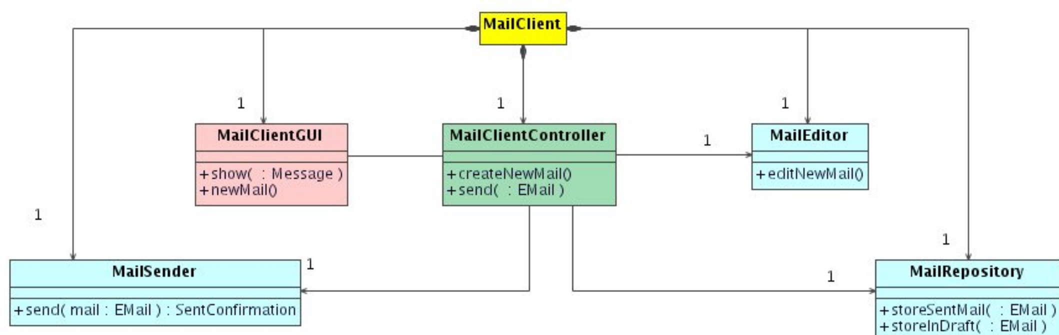
Figure 6. Communication diagram simplifying transition to collaboration context.



2.5. Projecting out the context of the collaboration

The collaboration shows the services the role players request from each other in the context of realizing the use case as well as the message paths required between them. URDAD now generates the context of the collaboration by projecting out the static structure required at the current level of granularity to realize the use case. This is a very simple step and the resultant class diagram is shown in Figure 7, “The context of the collaboration.”.

Figure 7. The context of the collaboration.



The context of the collaboration is thus that subset of the static model which, at the current level of granularity, is required to realize the use case. The objects from that level of granularity are peers and hence the relationships between them will be associations (client-server) and not aggregation or composition relationships (otherwise the objects would not all be at the same level of granularity). However, some of these are components of the higher level context, filling in composition relationships between layers of granularity.

Note that unlike many other design methodologies (see for example [Ben-Abdallah-et.al-2004]) which go from the use case model to the static model (often via an object dictionary), URDAD defines the dynamics realizing the use case first. Only then is the required static structure identified. URDAD thus generates minimal structural complexity, i.e. only those structural features which are

actually required to realize the use case.

Note also that the objects and classes generated are all at the same level of granularity. They are all either components of the use case context or actors. We have explicitly refrained from introducing any lower level classes at this stage. Instead URDAD provides a simple approach for going over to the next lower level of granularity.

2.6. Service provider contracts

Like many other modern design approaches, URDAD is also contract centric. For each component at any level of granularity one first specifies the contract. After all the difference between a class and a component is that the latter realizes a contract as specified by an interface with pre- and post conditions, invariance constraints and quality requirements. Contracts facilitate not only pluggability but also testability -- what would you be testing if there was no contract?

> **Note**
>
> Tests should be written for contracts (interfaces), not for classes. Any service provider claiming to realize a specific contract should be tested against the test for that contract.

URDAD requires that for each responsibility there should be a contract against which all service providers (components) which realize that responsibility should be tested. The contract may have functional aspects and non-functional aspects. The functional aspects are defined within the standard design-by-contract framework (see [Meyer-1991] and [Meyer-1992]) by an interface with pre- and post-conditions on the services and, if applicable, invariance constraints on the service provider itself. Contracts are developed from the perspective of the client. They thus resemble the “signed contracts” of Andreas Rausch (see [Rausch-2002]).

The non-functional aspects may include features like scaleability, usability, reliability, security, and so on. These are typically specified in a quality requirements note.

2.6.1. Pre-Conditions

If any of the pre-conditions is not met, the service provider is entitled to refuse the service without breaking the contract. On the other hand, if all preconditions are met, the service provider is obliged to provide the service. Otherwise it is a breach of contract and hence a failure. For example, for the debit service of an account there may be the pre-condition that there must be sufficient funds in the account. If there are insufficient funds the account may refuse the service without breaking the contract.

In software systems service providers use exceptions to notify clients that a requested service is refused due to a pre-condition violation.

2.6.2. Post-Conditions

The post-conditions are the deliverables of the service provider. These include the return value, but may also include service provider state information.

For example, the post-condition of debiting an account may include the requirement that the transaction must have been entered into the account's transaction history.

2.6.3. Invariance constraints

These are symmetry rules around the service provider's state. If at any stage (or at least on transactional boundaries) any of the invariance constraints are not met, then the object and hence the system is in an invalid state.

For example, for an account the invariance constraint could be that the sum of all credits minus the sum of all debits must always yield the current balance. If at any stage (at least on transactional boundaries) this symmetry does not hold, then the account and hence the system is in failure.

2.6.4. Quality requirements

From design by contract we know that the interface together with the pre- and post-conditions and invariance constraints provide a complete functional requirements specification for a service provider. In addition to the functional requirements, service provider may also need to adhere to certain non-functional (quality) requirements like scaleability, reliability or security requirements. Adding these quality requirements to the functional requirements completes the contract.

For our `MailSender` component we could require auditability as a non-functional requirement. The latter may be facilitated through logging all send requests together with their completion status.

2.6.5. Contracts and testing

Tests should be developed for contracts, not for individual service providers. The tests need to test both the functional, as well as the non-functional aspects of a contract.

During functional testing one tests that, if all preconditions are met, the service is provided such that

1. the client obtains the correct return value,
2. all post-conditions are met, and
3. if the invariance constraints were met prior to the service having been requested, that they are still met after the service has been provided.

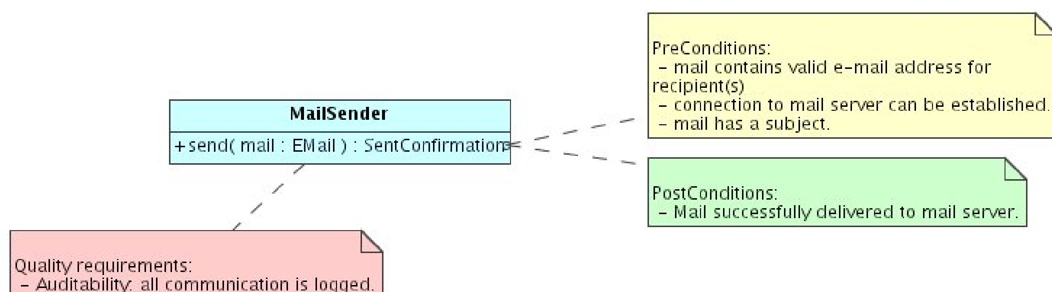
2.6.6. Contracts and the Object Constraint Language

UML enables one to specify the contracts formally in OCL, UML's *Object Constraint Language*. Doing this facilitates the automatic generation functional and system integrity tests.

2.6.7. MailSender contract

Figure 8, “Contracts are specified for each responsibility and hence for each service provider.” shows an informal contract for the `MailSender` component. It shows the required services together with the pre- and post-conditions for them as well as the non-functional auditability requirement. The contract could be formalized using the OCL.

Figure 8. Contracts are specified for each responsibility and hence for each service provider.



2.7. Value objects

One still needs to specify the structure of any object exchanged between system components or between system components and actors, the so called “value objects”. If we look at our mail client, then we see that an instance of an `EMail` is sent from the `MailClientController` to the

MailSender and that the latter returns an instance of a SentConfirmation. We need to specify the structure of these value objects. This is naturally done using UML class diagrams.

Figure 9. Class diagram for the e-mail value object.

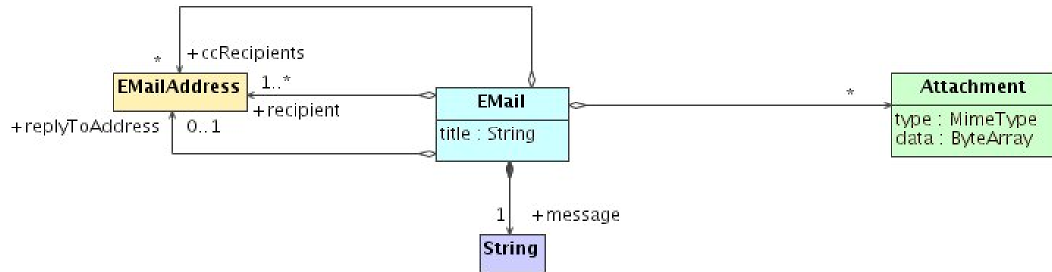
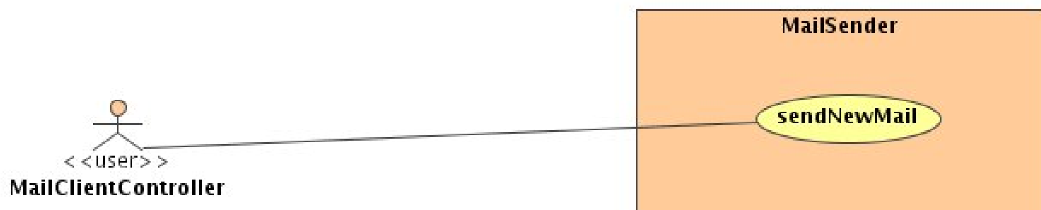


Figure 9, "Class diagram for the e-mail value object." shows a simplified class diagram for the e-mail value object exchanged between the MailEditor, MailClientController and MailSender.

2.8. Transition to the next level of granularity

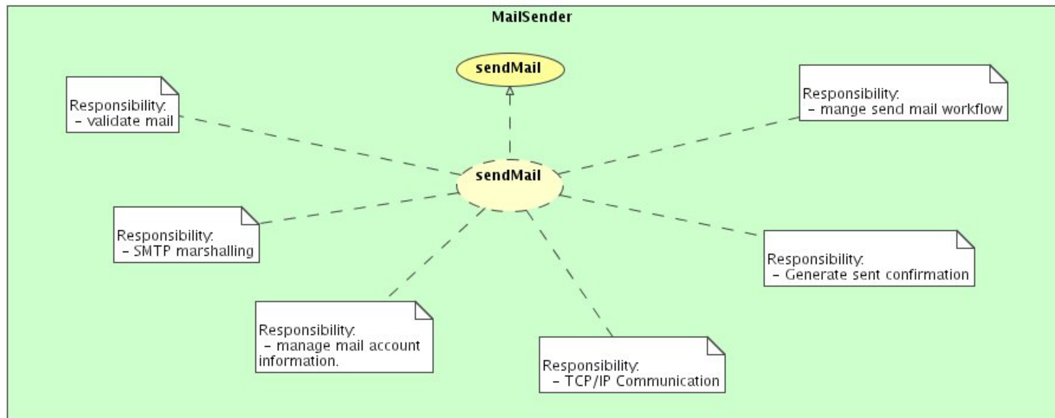
In order to go over to the next level of granularity, one selects one of the components from the current level of granularity as the context for the next level of granularity. Its services at the current level of granularity will become the use cases of the next level of granularity. Those components which interface with that object at the current level of granularity will become the actors. For example, if we select the MailSender as our new context, the corresponding use case diagram would be given by Figure 10, "Use case diagram for a component at the next lower level of granularity.", i.e. the MailSender is used by the MailClientController to send e-mails.

Figure 10. Use case diagram for a component at the next lower level of granularity.



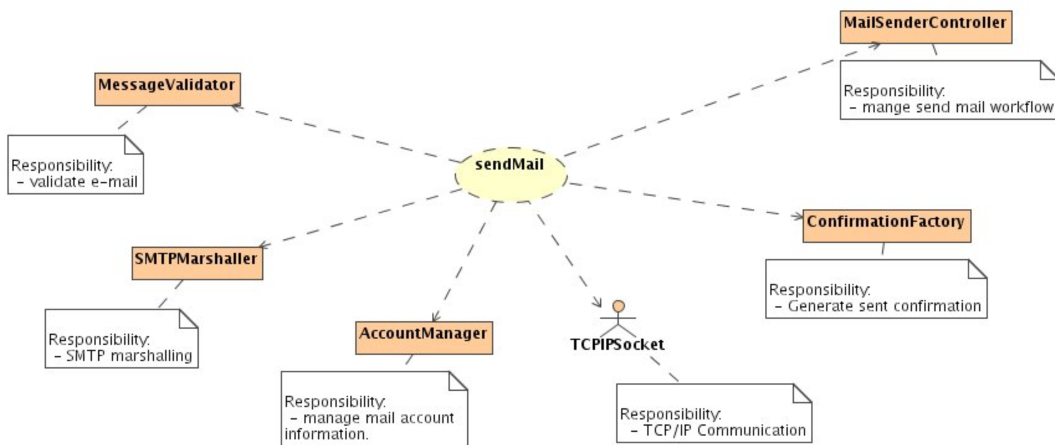
Of course the object may choose to realize the use case in a way which makes use of further actors. This should, however, be responsibility driven. We thus first identify the responsibilities at this new, lower level of granularity and before assigning them to components and/or actors of this lower level object as shown in figure Figure 11, "Responsibility identification at next lower level of granularity.".

Figure 11. Responsibility identification at next lower level of granularity.



Each responsibility is, once again, assigned to either a component of this lower level component (this is where the composition relationships are identified) or an actor. This is shown in Figure 12, “Responsibility allocation at next lower level of granularity.”

Figure 12. Responsibility allocation at next lower level of granularity.



Note that we have a workflow controller at each level of granularity taking over the responsibility of managing the workflow for the use case at that level of granularity Similarly, if the user interfaces directly with the lower level object (as, for example, the `MailEditor` would), then there would also be the responsibility of interfacing with the user.

URDAD is thus an iterative design process which projects out one level of granularity after another. Typically one will require between 2 and 4 levels of granularity depending on the complexity of the system.

3. Summary and conclusions

URDAD provides a simple algorithm for designing a use case realization. It projects out different levels of granularity. At each level of granularity one starts with responsibility identification followed by responsibility allocation. Once one has established the objects which take care of the responsibilities which need to be addressed when realizing a use case, one specifies how they collaborate. From the collaboration one can project out the collaboration context which is that subset of the static model which, at the current level of granularity, is required to realize the use case. URDAD then provides a simple mechanism for going from the current level of granularity to the next lower level of granularity. This is done by selecting one of the components from the higher level of granularity as new context. Its services at the higher level of granularity become the use cases at the lower level of granularity. Some of the components of the previous level of granularity

may become actors at this lower level of granularity. The process continues symmetrically by identifying and assigning responsibilities at the new lower level of granularity to components of the new context and external service providers (actors). The collaboration realizing the lower level use cases project out the static model at this lower level of granularity.

URDAD encourages *"good design"* by generating clean layers of granularity with minimal structural complexity, enforcing responsibility localization and contracts across system components.

Finally, URDAD provides a design process which follows the spirit of OMG's Model Driven Architecture (MDA) by generating a platform independent model (PIM). The PIM is then mapped onto some choice of architecture and technologies resulting in the platform specific model (PSM) which is ultimately mapped onto a realization resulting in the Enterprise Deployment Model (EDM).

Bibliography

[Ben-Abdallah-et.al-2004] H. Ben-Abdallah, N. Bouassida, F. Gargouri, and A. Ben-Hamadou. "A UML Based Framework Design Method". Journal of Object Technology. 3. 8. 2004.

[Frankel-2003] David S. Frankel. *Model Driven Architecture: Applying MDA to enterprise computing*. John Wiley & Sons. 2003.

[Meyer-1991] Bertrand Meyer. *Design by Contract*. Prentice Hall. 1991.

[Meyer-1992] Bertrand Meyer. "Applying Design by Contract". Computer (IEEE). 25. 10. October 1992. 40-51.

[Rausch-2002] Andreas Rausch. "Design by Contract + ComponentWare = Design by Signed Contract". Journal of Object Technology. 1. 3. 2002.

[Rosenberg-Scott-1999] Doug Rosenberg and Kendall Scott. *Use Case Driven Object Modeling with UML: A Practical Approach*. Addison-Wesley Professional. 1999.

[Wirfs-Brock-McKean-2002] Rebecca Wirfs-Brock and Alan McKean. *Object Design: Roles, Responsibilities and Collaborations*. Addison Wesley Professional. 2002.

[Wirfs-Brock-Wilkerson-1989] Rebecca Wirfs-Brock and Brian Wilkerson. "Object-Oriented Design: A Responsibility-Driven Approach". OOPSLA '89 Proceedings. 71-75. October 1-6, 1989.

[Wirfs-Brock-Wilkerson-Wiener-1990] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall. 1990.