NTNU – Trondheim
Norwegian University of
Science and Technology

Real-Time Scheduler Simulator

A Software Development Project in C++

Inger Johanne Rudolfsen

Master of Science in Cybernetics and Robotics
Submission date: June 2015
Supervisor: Sverre Hendseth, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Assignment

Development of a real-time scheduling simulator.

The student is to conduct a software development project by the rules of the art.

The student will develop a simulator of a real-time scheduler, well suited for testing different scheduling strategies on different task sets, generate relevant reports, statistics and figures. The motivation is to support upon the research of real-time schedulers. Thus, it is important for the simulator to be as open as possible for exploring existing and future scheduling strategies. Specifying requirements in cooperation with a potential user is considered a part of the project.

Theoretical part:

- Examine existing related simulators.
- Research the width of existing real-time scheduling strategies to generate requirements for the simulator.
- Research "best-practices" on relevant fields, such as software project development, software specifications, design methodologies, design patterns and documentation.

Deliverables: In addition to the project report, the software itself and related documentation shall be delivered as basis for the evaluation of the project.

Preface

This report presents and summarizes my MSc thesis in Enineering Cybernetics at the Norwegian University of Science and Technology. The work on this thesis was done during the spring of 2015.
The report is written for a person with some experience with real-time scheduling and the C++ programming language, and little or no experience with software developing methodology.

A sincere thanks goes to my supervisor, Sverre Hendseth, who has provided me with academic counseling as well as motivation.

June 8, 2015
NTNU, Trondheim

Inger Johanne Rudolfsen

Summary

In this project, a software called the `Scheduler Simulator` was created to simulate task executions given a specified real-time scheduler. Existing similar simulators were examined and evaluated, reusable functionality best fitted to be adopted by this projects simulator software was identified. Whereas all the existing solutions have strengths and weaknesses, the following properties were extracted: limit the end-users responsibilities, create thorough documentation, use Gantt charts with clearly marked deadline misses, and provide relevant and readable simulation results. Best practices related to software development, such as software development methods, documentation development and software design, have been researched. In addition, other relevant theory related to real-time systems and schedulers was also examined. To develop the Scheduler Simulator, the best practices were evaluated. For the `Scheduler Simulator` project, the development was conducted using mainly iterative and incremental methods, influenced by sprints from Scrum development, emphasizing the flexibility and simplicity described by Agile development methodologies.

The Scheduler Simulator allows a user to implement new scheduling policies, create and generate task sets and conduct simulations to generate relevant reports. The resulting reports contain CPU idle time, the number of reached and missed deadlines, the number of completed tasks, and a Gantt chart, visually presenting the simulated task executions. Four schedulers were implemented in the software: FIFO, RR, RM and EDF. A series of test runs of the Scheduler Simulator were conducted and evaluated. The simulator behaved as expected. It was discovered that when simulating more than 20 tasks, the Gantt chart overshot the page size of the result report. As a reaction to this, the simulator was altered to not produce Gantt charts for simulations where the task set exceeded 20 tasks. The documentation created for the system consists of three main parts; one for requirements, one describing the architectural design of the software, and one for the users. *"The users"* here refers to both maintenance personnel and end-users.

As the project was conducted by one person, the social benefits from using Scrum and other Agile development methodologies diminished greatly. Instead, the development work converged towards a waterfall shaped structure, as a sequential work model can be more appealing and natural when working alone than a parallel and cyclic model. The iterative way of working still helped the developer to learn from mistakes along the way and create a functional system.

Sammendrag

I dette prosjektet ble en programvare, kalt `Scheduler Simulator`, utviklet for å simulere prosesskjøringer i et system, gitt en sanntids-scheduler. Lignende eksisterende simulatorer ble undersøkt og evaluert for å kartlegge god funksjonalitet som kunne brukes i den nye simulatoren. De viktigste egenskapene som ble oppdaget ved de eksisterende løsningene var: å begrense sluttbrukernes ansvar, skape grundig dokumentasjon, bruke Gantt-diagrammer med tydelig markerte tidsfristbrudd og generere relevante og lesbare simuleringsresultater. Beste praksiser relatert til programmvare utvikling ble undersøkt, som programvareutviklingsmetoder, utvikling av dokumentasjon og programvaredesign. Annen relevant teori knyttet til sanntidssystemer og schedulere ble også undersøkt. For å utvikle simulatoren ble de beste praksisene innen programvareutvikling evaluert. Utviklingen ble gjennomført ved bruk av hovedsakelig iterative og inkrementelle metoder, med vekt på fleksibilitet og enkelhet som beskrevet av Agile utviklingsmetodikk.

Dokumentasjonen som ble laget for systemet består av tre hoveddeler; en for systemkrav, en for den arkitektoniske utformingen av programvaren, og en del for brukerne. *"Brukerne"* her er ment til å være både vedlikeholdspersonell og sluttbrukere.

Scheduler Simulatoren lar brukeren få implementere nye schedulere, definere og generere prosesssett og gjennomføre simuleringer som produserer relevante rapporter. De resulterende rapportene inneholder CPU inaktivitets tid, antall nådde og brutte tidsfrister, antall fullførte prosesser, og et Gantt-diagram, som gir en visuell presentasjon av de simulerte prosessene. Fire schedulere ble implementert i programvaren: FIFO, RR, RM og EDF. En rekke testkjøringer av simulatoren ble utført og evaluert basert på de genererte rapportene. Simulatoren oppførte seg som forventet. Det ble oppdaget at da man simulerte mer enn 20 prosesser, overgikk Gantt-diagrammet sidestørrelsen i resultatrapporten. Som en reaksjon på dette, ble simulatoren endret til å ikke produsere Gantt-diagrammer for simuleringer der antallet prosesser var mer enn 20.

Ettersom prosjektet ble utført av én person, ble de sosiale fordelene ved å bruke Scrum og andre Agile metoder redusert betraktelig. Utviklingsarbeidet konvergerte i stedet mot en fossefall formet struktur, ettersom en sekvensiell arbeidsmodell kan falle mer naturlig når man arbeider alene enn en parallell og syklisk modell. Den iterative måten å jobbe på hjalp likevel utvikleren til å lære av sine feil underveis i utviklingen, og til å skape en funksjonell simulator.

Contents

Chapter 1

Introduction

Intelligence and rational thinking are human traits adapted by technology. By transferring the human way of rational thinking to technological algorithms, software systems can reach decisions using computerized logic. The number of operations computers can conduct continues to increase, leading to a continuous expansion of the world of technology.

1.1 A Modern Glimpse of the Future: Scheduling in Real Time Systems

Efficient computer systems develop rapidly, finding smarter and better solutions faster every day. Optimizing the logic in computer systems grows more important as the technological evolution demands faster and more efficient systems. A scheduler decides when to execute which tasks. Choosing which scheduler to use for a system can be crucial for its overall performance.
We rely on real-time systems. Everyday chores, like driving a car, using a dishwasher, not to mention smartphones, are all real-time systems. It is important that these systems are optimal to maintain a productive and operative society. To achieve this, the real-time tasks must be executed in an optimal fashion, which may require a glimpse of the future. A scheduler can provide us with this "glimpse", by using a scheduling algorithm to control the task executions in a system. Scheduling research continues to optimize the way real-time systems operate, and might help to ensure faster and better technology in the future. Knowing what needs to be done is no longer sufficient. Knowing how to choose when to do what is key.

1.2 Exploring the Assignment

For the assignment, a simulator software is to be developed. Simulating the behavior of real-time schedulers can help optimizing scheduling policies, and avoid deadline misses in real-time systems. The supervisor of this project had a vision of what properties the simulator software should possess. He thus took the role as the *"client"* in the development project, while the author of this report was the *"developer"*.
The term *"best-practices"* is interpreted to be the commonly used business term for a practice that induce superior performance of some sort compared to practices within the same field. A practice can also fall under the term *"best-practice"* if it is widely used and considered a standard procedure within several organizations.

1.3 Structure of the Report

This report is structured to promote and convey the project progression and results. Since the use of software development methods is essential in this project, the report presents not only the resulting software, but the transpired development progress as well, as it displays how software development methods influenced the workflow. Chapter four is dedicated specifically to describe the work and development process, while chapter six contains the description and documentation of the final software product.
The report is structured in the following way:

Chapter 1 is the introduction (which you hopefully are aware that you are currently reading).

Chapter 2 contains background theory related to software development, documentation and software design.

Chapter 3 contains background theory related to real-time systems and scheduling, presentations of existing scheduler simulators and descriptions of tools and software used for the development of the new simulator software.

Chapter 4 contains an evaluation of the existing solutions and a description of the development process, including iterative progress, design choices and test runs of the system.

Chapter 5 displays the results from the test runs of the simulator described in chapter 4.

Chapter 6 contains the software documentation, divided in three main parts: *requirements, architecture and design,* and *user documentation.*

Chapter 7 is the discussion, which addresses and evaluates key points from this project, such as the software development methods, the final software, the documentation, etc.

Chapter 8 is the conclusion, consisting of a summarizing conclusion of the thesis, and a section containing recommendations for further work, describing ways to improve the created `Scheduler Simulator` software.

Chapter 2

Background Theory: Software and Design

2.1 Software Development Methods

Developing a software program can be a time consuming business. Scheduling and planning the work process is thus very important to reach finalization of the product within a time frame. There exist different methods describing the work process using guidelines and structure to improve productivity and workflow. These methods are called *Software Development Methods* and can be used when developing a software. Some of these methods are applicable to other developing projects not involving software, and are broadly used in different fields. The methods are often developed over many years, and have known strength and weaknesses. Even though a method works for one project, does not necessarily make it suitable for other projects. It is therefore common practice to "pick-and-mix" strategies and models from the different methodologies to accommodate a specific project. *"Best practices"* within software development refers to some methods and models, not because they can be marked as being the "best", but because they are commonly and largely used for software development, (Hasle, 2008).

2.1.1 Waterfall Model

The Waterfall Model is a strictly structured system development model, built up by consecutive phases, describing the development process. When one phase is completed, the next phase begins, and thus the development progress flows through the phases, like a *waterfall*, see figure 2.1.The phases of the waterfall model exists in many variations, but according to Hasle (2008), they are:

- **Pre-analysis:** Specifying requirements.

- **System analysis:** Detailing the system requirements.
- **Construction:** Constructing objects, designing architecture.
- **Implementation:** Coding and testing the system.
- **Operation:** Project finishes. Ownership and responsibility is transmitted.

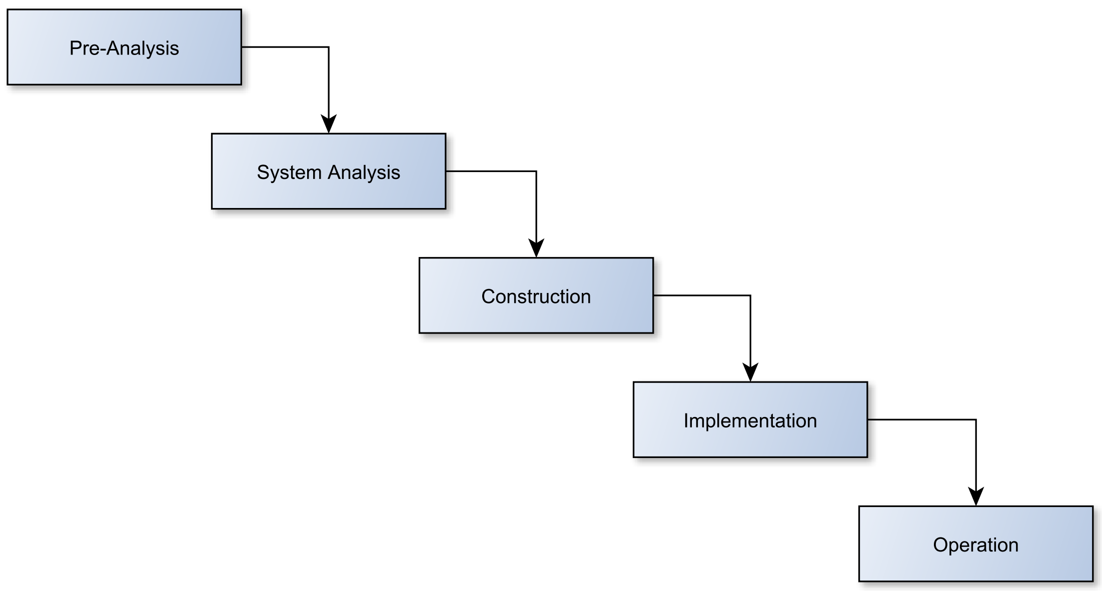


Figure 2.1: The waterfall model, based on figure 5.1 in Hasle (2008).

A phase is completed only when it is properly reviewed and verified. If a phase cannot be completed because of major mistakes committed in previous work, some variations of the waterfall model allows jumps back to previous phases, to attempt to correct the earlier flaws. The firm structure of the waterfall model makes it understandable and clean, but it also causes problems. In real life, this linear way of progression is rarely possible.
The waterfall model can be used to avoid expensive costs. By spending time building the system and software requirements, one may discover a flaw in the software. It is much cheaper to correct this flaw early, than to fix a major bug later in the coding phase.
Developing using the waterfall model doesn't produce a working software until the end of the development process. If working with clients, they have to specify requirements, wait for the software and then, at the end of development, experience the finished software for the first time. The finished software may not be exactly

what the client had in mind, and thus the requirements have to change, and the development has to start from the beginning, at the top of the waterfall model. Such an occurrence can be very expensive. The model is thus more suitable for project with fixed requirements and understandable technology, (Hasle, 2008). The `Sashimi` model, presented in the book McConnel (1996), is an alteration of the Waterfall model, with overlapping phases, emphasizing that important useful information can be acquired in phase transitions.

V-Model

The *V-model* is considered to be an extension of the waterfall model, using many of the same phases and the same flow-inspired progression. The V-model associates each phase related to developing to a test phase, creating a V-shaped model, see figure 2.2. In every development phase, a test is created for this particular phase. Planning and creating these tests before coding helps to prevent flaws, and to ensure that phase goals are met. The V-model is not very flexible, and is, like the waterfall model, best suited for projects with fixed requirements. Also here, no prototypes are produced, and there exists a risk that the finished product might not meet the clients expectations (Balaji, 2012).

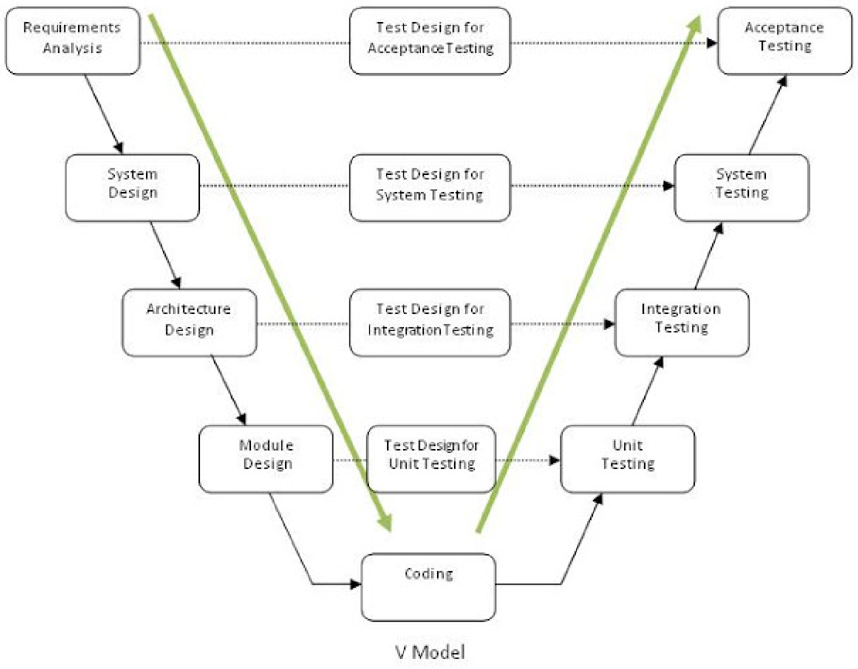


Figure 2.2: The V-model, (Balaji, 2012).

2.1.2 Iterative and Incremental Development

"Practice makes perfect" is a known saying, with the underlying meaning that in order to succeed, one must try, learn and try again. This is the essence of *iterative and incremental development.* Starting with a small part of the wanted software, requirements are gathered, an analysis is conducted, the code is implemented, tested and evaluated. This cycle is repeated if needed, allowing the software developers to learn from their mistakes. Piece by piece, the software is developed. The cyclic behavior model is called *an iterative design*, meant to improve code quality and design. Starting with a small part of the software and gradually expanding is called an *incremental build model.* Usage of these methodologies gives flexibility in code development, and provides parts of the software in working code at an early stage. Iterative and incremental development methods are essential in many other software development methods, such as Agile Software Development Methodology, section 2.1.7.

2.1.3 Software Prototyping

A prototype is an early sample version of a product. The prototype can be incomplete, or lack great functionality intended for the final product. Meeting a clients expectations can be difficult, and so prototyping early drafts of the software program, and have the client evaluate them can be helpful to create the product the client wants. The flow of a prototyping project can be seen in figure 2.3:

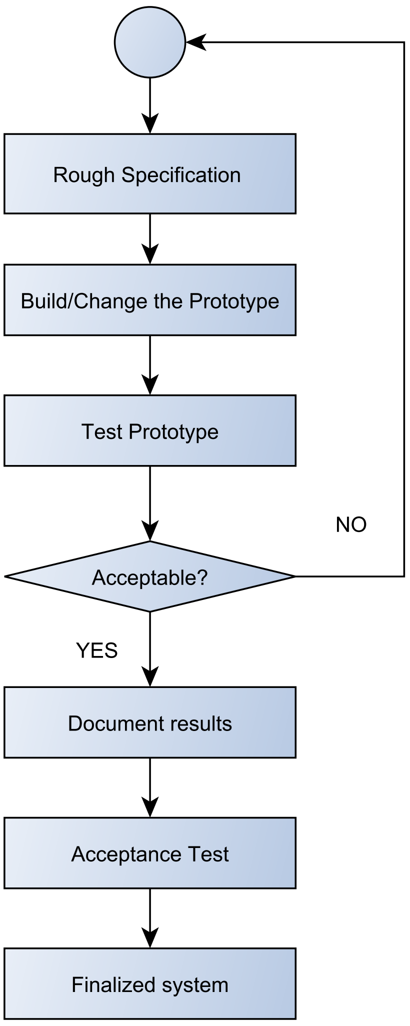


Figure 2.3: The Prototyping Flow, based on figure 7.4 in Hasle (2008).

Prototyping can reduce cost. Not meeting the clients expectations with a finished product can be very expensive. Used wrongly, prototyping can cause costs when, for example a developer gets too attached to a prototype's features. A prototype is intended to be thrown away, and spending too much time developing one prototype can be expensive. Prototyping can be used in various degrees in any project, but is best suitable for projects developing a system dependent on user interactions.

2.1.4 Rapid Application Development

As mentioned in section 2.1.1, the waterfall model rigorously maps out the requirements before proceeding. This is not always beneficial, and *Rapid Application Development, **RAD***, is an alternative to the waterfall model, emphasizing speed and quick development processes. Flexibility and quick development is central in RAD, and rather than spending time specifying the requirements, development is started early.
RAD is divided in four phases (Hasle, 2008):

- Requirements planning : Planning the project and stating requirements
- Prototyping: see section 2.1.3.
- Testing: As the prototyping phase takes care of initial testing, this phase is beta-testing in the clients environment and final acceptance testing.
- Operational phase: The product startup, commissioning.

RAD is especially effective for developing services driven by user interfaces, as changes in requirements are allowed by the emphasized flexibility, but, as a trade-off, an RAD process can be difficult to control.

2.1.5 Spiral Model

Th *spiral model* is a generated using elements from other models, like the previously mentioned waterfall model in section 2.1.1, incremental development in section 2.1.2 and prototyping in section 2.1.3. The generation of the model is *risk-driven*, thus the risk patter of a project determine how the spiral model should be used. The combinations of many models gives the model a spiral look, shown in figure 2.4. The goal is to harvest benefits from other models, and avoid the pitfalls.

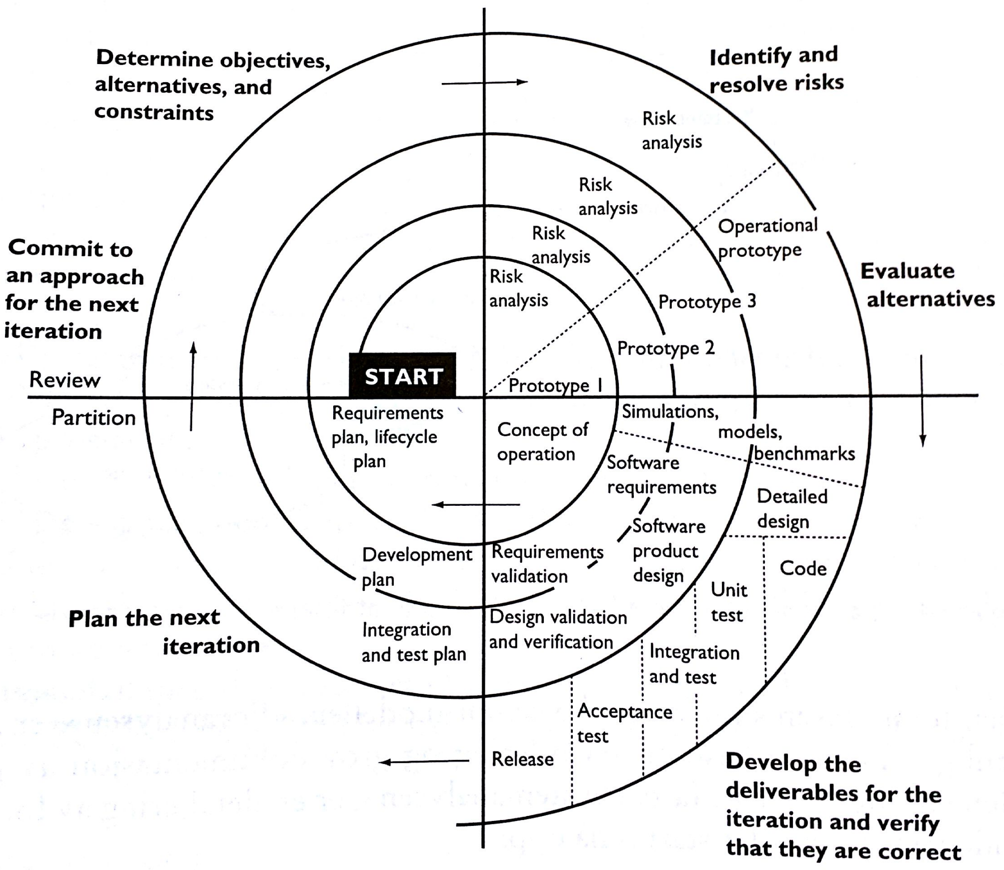


Figure 2.4: Boehms spiral model, from figure 5.5 in(Hasle, 2008).

2.1.6 Test-Driven Development

Test-Driven Development, **TDD**, as explained by Grenning (2011), is an iterative development technique, used to avoid large debugging sessions, and rather ensure that the created code executes correctly. In TDD, before creating a new feature, a *unit test* is developed. This test is made to verify that the requirements for the new feature are met, and it should fail otherwise. When a test is developed, some code is written to pass the test. This code is rapidly made with a single goal: to pass the test. When this is done, the initial code is refactored and cleaned. For each new feature, this cycle is repeated. Writing the tests *before*, and not after, writing the code, helps the developer keep the requirements in mind, and avoid massive debugging. Failing a test indicates a bug in the code, and enables the developer to fix the bug right away, and not later when the entire code is finished and time has passed. This also allows the developer to code, compile and re-code a small part of a software in a short period of time, compared to the long time it would take to compile, code and recompile a large software. The tests are also used to ensure that new code doesn't make old features fail their unit tests. TDD focuses on functionality, and can result in clean, simple code and design. If a unit test case is falsely letting unfit code pass, the tests may cause a false feeling of success, and induce extra maintenance issues later the project.

2.1.7 Agile Software Development

Agile Software Development methodology is a collective term, referencing to software development methods that emphasizes flexibility, adaptive planning, evolutionary development, continuous improvement and early delivery. Developed from incremental method, *lightweight* methods arose in the 1990s as a reaction to the *heavyweight* method, such as the waterfall model, section 2.1.1. The term *Agile* was not used before 2001, when a group of 17 developers created the *Manifesto for Agile Software Development*, (Cockburn, 2001). Even though many of the *lightweight* methods predated this manifesto, they are referred to as Agile methods today, because of their adaptive and flexible nature. For developers working together using Agile development, teamwork and collaboration is important in throughout the project. Implementing agile development methods require a flexibility and maneuverability, and a large team of 40 developers make this near to impossible. Smaller teams, of no more than eight people, are preferable, (Cockburn, 2001). Slogans like *"Model with a Purpose", "Travel Light"*, and *"Working Software is The Primary goal"* arose from the Agile way of thinking, indicating that sufficient documentation and little bureaucracy makes the wheels of development turn (Hasle (2008) and Shore (2007)).

"Working software over comprehensive documentation."
- Agile Manifesto (Cockburn, 2001).

2.1.8 Scrum

Opposed to a sequential, micro-managing approach to create a software, *Scrum* is meant to connect a team of developers to reach a common goal, maximizing the teams abilities to do so. The team consists of three main parts: *the product owner, the developing team and the scrum master*. The *product owner* ensures that the produced product is value added, and she functions as a communicative bridge between the team, the stockholders and the costumers. The product owner may also be part of the development team. *The development team* is responsible for the actual development work, like analyzing, designing, coding and testing. *The scrum master* ensures that the scrum process is being followed and used as intended.
A *sprint* is the unit of development in scrum. A sprint is restricted to a specific

duration, a *time box.Timeboxing* is the strategy of using time, and not functionality, to determine the length of a sprint, (Hasle, 2008). If the functionality goals are finished prior to the timebox's duration, additional functionality or expansions are added to the sprint. A sprint is built up by three main parts: *the sprint planning, the daily scrum and the sprint review.*
The sprint planning is at the beginning of a sprint, mapping the goals for the sprint, setting a timebox, and prepare the sprint *backlog.* The backlog consists of whatever needed to be done in order to successfully deliver a viable product. The sprint backlog is made from unfinished or improvable work from previous backlogs. The backlog is a kind of requirement specification, which makes up the workload for the next sprint.
The daily scrum is a fifteen minutes long daily meeting for the scrum team, where all the members of the team have to answer three questions:

- What did I do yesterday that helped the development team meet the sprint goal?
- What will I do today to help the development team meet the sprint goal?
- Do I see any impediment that prevents me or the development team from meeting the sprint goal?

Answering these questions helps developers reflect on their work, and can shed light on problems or issues that should be addressed.
The sprint review and *sprint retrospective* occur at the end of a sprint. The sprint review is when the team review the work that has been done and present the complete work to the stakeholders. The sprint retrospective is when the team reflects on the passed sprint, and agrees on actions to improve as a team. (Hasle (2008) and Cockburn (2001))

2.1.9 Extreme Programming

*Extreme Programming, **XP**,* is a socially directed development method, inspiring to create a healthy work environment, where productivity and quality are emphasized. Being an Agile method, XP uses frequent releases and checkpoints to meet the customers requirements. XP is a method used to improve software quality, and is based on five *values* (Beck, 2005):

- Communication : A flow of information prevents potential problems.
- Simplicity: Rather add something tomorrow than to create overly complicated code today.

- Feedback: Tests provide instant feedback.
- Courage: Dear to make decisions. Deal with problems when they occur, not later.
- Respect: The foundation of XP and teamwork.

It is important that the developers embrace these values when conducting the four basic *activities* in XP development: *coding, testing (often using TDD, section 2.1.6), listening to the customers* and *designing*. Developers must be open minded to change, and can not get to attached to a product. If a customer changes their requirements drastically, a developer must embrace the change, and not try to work against it, holding on to old code. In XP development, it is common to use *pair programming*, when two developers shares a workstation. *The driver* writes the code, while the other one, *the observer* reviews the code as it is typed. This helps avoiding flaws and code-errors. The driver and observer switch roles frequently.
"You aren't gonna need it", **YAGNI**, is a principle for XP development, meaning that you simply create code when you need the functionality, and not add functionality for foreseen scenarios. This promotes the value of simplicity in XP. Continuous integration is important, meaning the developer should not sit hours on working with the same code without integrating it with the rest of the system. This is to avoid collisions with already existing code (Beck (2005) and Hasle (2008)).

2.1.10 Other Development Strategies

There exists a vast number of different development methodologies and philosophies. The most popular and widely known are in the above sections. Some other strategies and principles used in software development are listed below:

- **Refactoring** can be used to increase the quality and internal structure of a software program, (Booch, 2000). Enables the developer to go back and improve the system without changing external requirements.
- **Cowboy Coding:** Start producing code without a design or a developing method.
- **Chaos Model:** Always resolve the most important issue first.
- **Worse is better:** The idea that quality does not necessarily increase with functionality. Less functionality can be a preferable option.
- **KISS:** Keep It Simple Stupid. Simplicity is often associated with good design. Do not over-think when developing.

- **"There's more than one way to do it"**, or **TIMTOWTDI**, is a strategy from PERL programming, meaning there are more than one way of solving a problem, providing freedom for the programmer. An extension of this strategy emphasizes the fact that some solutions are more common simply because they are better, resulting in the neat expression: *"There's more than one way to do it, but sometimes consistency is not a bad thing either"*, or **TIMTOWTDIBSCINABTE**, pronounced *Tim Toady Bicarbonate.*

2.2 Documentation Methods

When developing a software program, it is important to remember decisions and choices made during the development to benefit from in later stages in the life time of the software. According to Hasle (2008), there are two main distinctions in documentation: *System documentation* and *User documentation.* System documentation aims to describe the software system with the intention the technical aspects of the system is presented. The system documentation includes requirement documentation, design, architecture and code documentation.

A developer is often creating a software for a customer. The user documentation must accommodate both end-users and maintenance personnel. It is important that sufficient documentation exists for an end-user, with no prior knowledge to the software, so she or he can use the software program as it was intended by the developer. This is called the *end-user documentation*, and it describes the how to use the system features. *Final-documentation* can be seen as a kind of user documentation, and it is primarily meant to be used for maintenance purposes (Hasle, 2008). The final documentation must give a correct and in-dept description of the system, sufficient for maintenance personnel to operate and possibly expand the software.

It is important to use a language understandable for the intended reader when writing documentation. Complicated words and technical terms serves against their purpose if used in a user-documentation meant to be read by a user with no prior experience with software development. Jerry Weinberg says:

"The value of documentation is only to be realized if the documentation is well done. If it is poorly done, it will be worse than no documentation at all."(Rüping, 2003).

This indicates that the documentation must serve it's purpose of communicating information. In the book Cockburn (2001), the author recommends documentation to be "light but sufficient". This corresponds to the Agile manifesto, stating "Working

software over comprehensive documentation", see section 2.1.7. The book Rüping (2003) lists solutions to common documentation problems, and recommends that for each project document, one person must accept responsibility. Further, it states that documentation must evolve as the project evolves, being "living" documents. Writing documentation, done correctly, can make the writer reflect on what they are writing and might help them discover alternate solutions for their projects, improving the final results. The kind of documentation needed varies from project to project, but a rule of thumb can be:
"The correct amount of documentation is exactly that needed for the receiver to make her next move in the game", (Cockburn, 2001).

2.2.1 Specifications and Requirements

Requirements specifies what the software is supposed to do. They can for example be strict constraints within the time domain, or wanted specifications for the user interface. They describe operations the software must be able to perform. A requirement *specification* is a set of requirements to be satisfied by the software service. The requirements works as a written agreement of the traits of the final product between all involved parts. Thus the requirements should be understandable for both users and developers. This causes a challenge to create a complete requirement documentation, because it should describe the in dept software constraints and behavior, and still be understandable for all parts involved. The requirement documentation is therefore often created by a combination of written text and models. The amount of requirement documentation needed depends on the nature of the project: A project where the resulting product may impact human lives, such as medical equipment and medical software, the requirements must be very detailed, as opposed to a short-life software prototype, which requires a minimum amount of requirements. There are different ways of eliciting requirements from the clients or potential users. Observing the potential users, or interviewing the clients are the most common ways of doing so. When interviewing, it is important to use a natural language, ask open ended questions, and simply let the client elaborate by asking the simple question *"Why?"*, (Wiegers, 2003).

Written Requirement Documentation

One way to document requirements is written sentences, stating what the software product should be able to do. Using general language, describing the agreed upon behavior in a written document ought to be understandable for both users and developers. Using complete sentences, starting with "The system must..." or "The user shall.. ", completed with behavioral description, can be a good way of painting a picture of the system. Ambiguous terms should be avoided, as they can cause confusion and wrong interpretations and expectations for the software. Important parts of the system must be stated clearly. (Wiegers, 2003)

Applying Context to the Requirements using UML

Unified Modeling Language, UML, is a unified way of communicating the structure and functionality of complex systems, (Douglass, 2004). UML consists of several ways of representing the expected behavior of a system, based on specified requirements. One of this methods is called a *Use-case*.

*"A **Use-case** represents a series of interactions between an outside entity and the system, which ends providing business value." (Guiney, 2003).*

The outside entity is often referred to as the *actor*. *An actor* can be a human, time or an external system. There is no strict set of rules when creating a use case, but there exists optional templates on what a Use Case can include. New templates can be created and altered when needed. An example of such a template is shown in table 2.1. A use case describes different types of scenarios an actor encounters when interacting with the system.

Table 2.1: A Use-Case Template with explanations, based on figure 2.13 in Guiney (2003)

Use-Case Name:	Descriptive title of the interaction presented in this use case.
Iteration:	Refers to the different steps of descriptive dept the Use-Case go through: *Facade*, which is a high-level description, *filled*, which is broadening and deepening, and *Focused*: which is the final iteration, narrowing and pruning.
Summary:	Two or Three sentences describing the outlines of the interaction.
Basic Course of Events:	The path of events to happen in the most common scenario when no errors occur.
Alternative Paths:	Describes alternative ways to reach the same outcome as the basic course of events.
Exception Paths:	Presents the course of events when an error occurs.
Extension Points:	References to other related Use-Cases.
Trigger	The criteria for entering the Use-Case situation.
Assumptions:	Dependencies that are assumed true for the Use-Case to be carried out.
Preconditions:	Conditions that must be in place prior to the Use-Case described interaction.
Postconditions:	Conditions arising in the aftermath of the Use-Case.
Author:	The Use-Case creators name.
Date:	The dates corresponding to the completion of the three development iterations of the Use-Case.

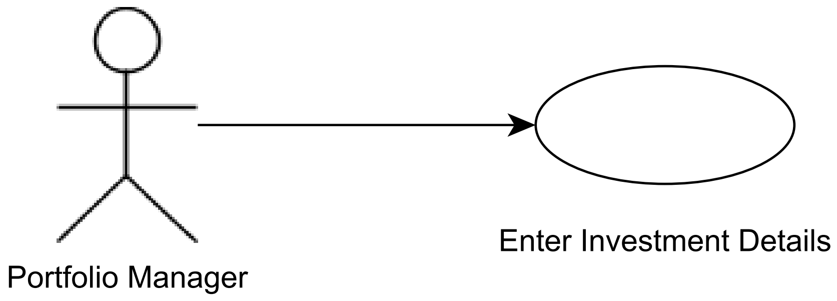


Figure 2.5: A Use-Case Diagram, based on figure 2.1 from Guiney (2003).

A *use-case diagram* is a visual representation of a relationship between an actor and a use-case. The actor is portrayed by a stick-figure, and an arrow points from the actor to the use-case goal. An example is shown in figure 2.5.

Other diagrams used to describe requirements are i.a. *activity diagram, Sequence Diagram, Flow charts, State machine diagram*(Douglass, 2004). Diagrams and Use Cases can be understandable for non-developers if a general language is used, but for complex systems, this can be difficult to achieve. The requirements involving algorithms, timing and safety-aspects are not easily portrayed this way, because these requirements are not based on interactions.

User Stories are closely related to use-cases, but they are shorter, and angled to show how the system can be used by different *roles* in a specific scenario. User Stories can often be written by the costumer, capturing their wishes for the product in writing. Cohn (2004) describes three aspects of user stories:

- Cards: A short written description of the story.
- Conversation: Conversations to fill out the details of the story.
- Confirmation: Using test to confirm when a story is completed.

The written User Stories are short and narrative. They can be vague, and thus not a precise way to map requirements. User Stories are mostly used as a check-up documentation during development to map progress, and are rarely regarded as full-descriptions of the requirements.

2.2.2 Architecture and Design

A *Software Design Document* is a written description of the software product, where the designer intends to guide developers through the architecture of the software.

The software design must be described using natural, unambiguous language. It is preferable to repeat some parts rather than leaving details out. *"A picture is worth a thousand words"* is a known saying. Using other tools than text description can be highly rewarding.

"Diagrams can provide exellent overviews, while an accompaning text explains details to the extent that is necessary", (Rüping, 2003).

Different diagrams can be used to illustrate the design and architecture of complex software systems.

A **class diagram** displays the relations between classes in a software program, and is esspessialy useful when creating object-orientated software, in such languages as C++. There exists different ways to create class diagrams. OMT (Object-Modeling Technique) was widely used in the mid 90-s (Gamma, Helm, Johnson, Vlissides, 1995), but in recent times, UML, see section 2.2.1, has become the most commonly used standardization. Each class is illustrated with a rectangle with three compartments. The name of the class is displayed in bold letters in the top compartment. The second compartment lists the class data members, and the third compartment lists the class functions/methods. A prefix describes the protection level of each entry, see table 2.2.

Table 2.2: Marks indicating visibility type

+	Public
−	Private
#	Protected

The relationship between classes are illustrated using different types of edges. The different edges are displayed in figure 2.6, 2.7 and 2.8 (Ambler, 2001).

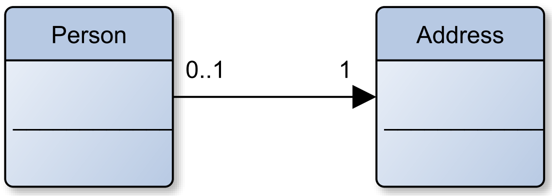


Figure 2.6: One class, Person, **contains** an object from another class, Address, as a data member.

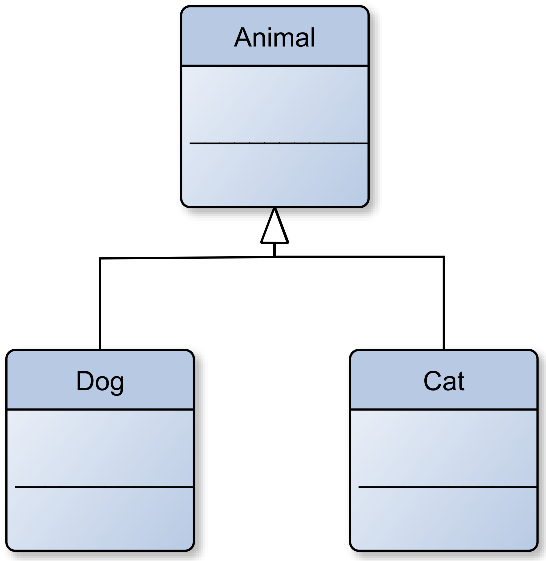


Figure 2.7: Two subclasses, Cat and Dog, **inherits** Animal class

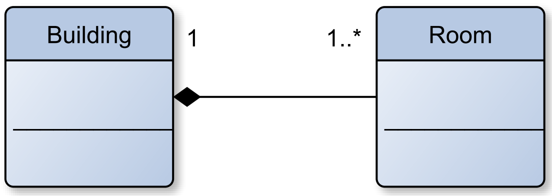


Figure 2.8: A class, Building, is **built from** objects of another class, Room.

2.2.3 User Documentation

The user documentation can be separated in two main parts: the *end-user documentation*, explaining features meant for the end-user of the system, and the *final documentation*, providing more advanced users and maintenance personnel an in dept description of the software.

Final Documentation

Final documentation is a document describing a technical product that is either under development or in use (Hasle, 2008). The goal of the documentation is for users and service personnel to fully understand the technical aspects, architecture

and functionality of the final product. Both requirement specifications, section 2.2.1, and design and architectural documents, section 2.2.2, can be a part of the technical documentation. In software development, code-documentation provides understanding of the software products features and it enables expansion and reuse of the code in the future. There exists a vast number of tools for generating code-documentation from within the source code (Rüping, 2003). *Doxygen* is a commonly used software tool for this type of documentation generation (van Heesch, 2006). This simplifies the developers work, as both code and documentation can be written at the same time, and it makes sure that the documentation automatically is up to date.
Without proper final documentation, the software dies when the developing team becomes unreachable due to time.

End-User Documentation

What is the use of developing a software program if nobody knows how to use it? Fairly little use. End-User Documentation describes how the product is to be used. The documentation's goal is for a user, with no prior knowledge of the product to be able to use the product in the intended way. This can be done in various ways, relying on readability and good structure. According to Rüping (2003), readability can be enhanced using a set of techniques, such as placing diagrams and tables close to their reference in the text, emphasizing words and sentences, setting line spacing to be 120% of the type size, and providing the reader with an brief overview of the content in the beginning of the document, describing the purposes of the different paragraphs. For software programs, *tutorials* are widely used to help new users get started. A *tutorial* is a step-by-step guide often consisting of simple introductory tasks the user has to complete. A correct constructed tutorial will give the user the tools to understand the basics of the program, and help him or her to apply the knowledge from the tutorial to the entire software system. A tutorial can be interactive, where the user contributes by f.e. solving tasks, or descriptive, showing how to use the program from the users point-of-view. The main goal of the end-user documentation stays the same: it must be easy to read and communicate the necessary information for proper usage of the product to the user.

2.3 Design Patterns and Principles

Software architecture describes the connections and interfaces between modules, classes and other components in a software program. The code-design can decide if the software is agreeable and easy to work with, or messy and difficult. The following four signs of bad architecture and design can show if the software is rotting from the inside out, according to Martin (2000):

Rigidity: The software is difficult to change without causing massive domino-effects. A change in one module will cause changes in various sub-modules, and what was supposed to be a small fix becomes weeks of work.

Fragility: Related to rigidity. A change in one module causes multiple breakages in various places of the code. The code can fail in places with no conceptual relationship to the original changed module. A small change in one place can cause an all-around code-fix.

Immobility: The inability to reuse software. Often, an engineer discovers the need for a module similar to another engineers module, but instead of changing the original module to be reused, she rewrites it.

Viscosity: When the design causes design-preserving solutions to be harder to implement than other solutions, the viscosity of design is high. When engineers choose a non-optimal solution that do not force recompilation because compilation-time is so long, the viscosity of environment is high.
(Martin, 2000)
These four symptoms are important to avoid when creating a software. Some problematic scenarios are more recurring than others when developing a software design, and there exists general solutions for these scenarios called *design patters.* These are *reusable*, well designed solutions for similar scenarios, developed through experience. A design pattern has four essential elements (Gamma, Helm, Johnson, Vlissides, 1995):

The pattern name, which describes the design problem and solution in one or two words.

The applicability, in other words, when to apply this pattern.

The solution, the design that solves the problem.

The consequences, the results and trade-offs by applying this pattern.

"*The gang of four*" are a well known group of developers that created a book about design patterns (Gamma, Helm, Johnson, Vlissides, 1995). In this book, patterns can be divided in three categories: *creational, structural* and *behavioral.* 23 patterns are described in this book, which has been *"... highly influential to* *the field of software engineering and is regarded as an important source for object-oriented design theory and practice."*(Design Patterns, Wikipedia, 2015). Some of the patterns are described in the following sections.

2.3.1 Creational Patterns

Creational patterns are patterns used to abstract the process of initiating, creating, objects.

Pattern name: Builder

Intent: *"Separate the construction of a complex object from its representation so that the same construction process can create different representations." (Gamma, Helm, Johnson, Vlissides, 1995)*
Applicability: The Builder pattern should be used when the assembling of parts that make up an object is isolated from the algorithms that create one or more complex, independent objects.
Solution: The structure of this pattern can be seen in figure 2.9. The client creates a Director object along with a desired Builder object, and calls the Directors Construct() function.

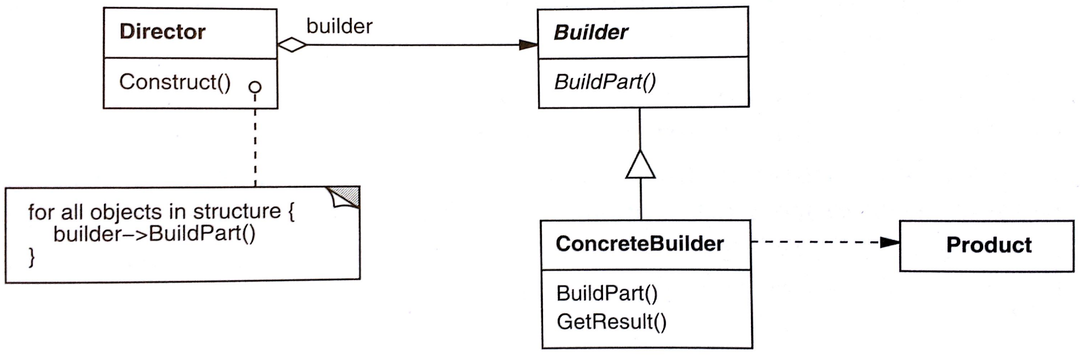


Figure 2.9: Structure of Builder pattern, (Gamma, Helm, Johnson, Vlissides, 1995)[page 98].

Consequences: see table 2.3.

Table 2.3: Consequences of the Builder pattern

Positive consequences	**Negative consequences**
Can vary a product's internal representation	May cause a large number of classes
Isolates code for construction and representation	
Gives a fine control over the construction process	

2.3.2 Structural Patterns

Structural patterns describe how larger structures can be created by combining classes and objects.

Pattern name: Facade

Intent: *"Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use." (Gamma, Helm, Johnson, Vlissides, 1995)*
Applicability: The Facade pattern can be used to create a simple interface to a complicated subsystem for a client to use. The pattern can also be used to layer subsystems.
Solution: The structure of this pattern can be seen in figure 2.10. Clients communicate with the subsystem through the Facade class.

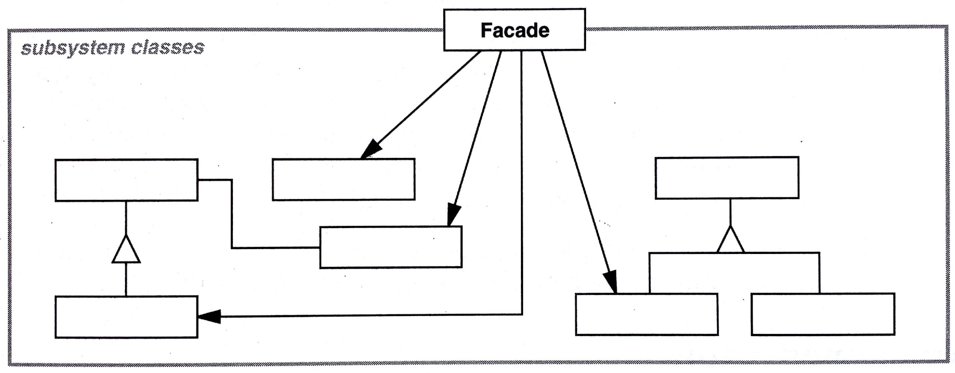


Figure 2.10: Structure of Facade pattern, (Gamma, Helm, Johnson, Vlissides, 1995)[page 187].

Consequences: see table 2.4.

Table 2.4: Consequences of the Facade pattern

Positive consequences	**Negative consequences**
The Facade class hides the subsystems components from the client, reducing the number of objects clients deal with.	The Facade provides a simple default way of using the subsystem that may not be broad enough for some clients.
The subsystem can be changed without affecting the client, by altering the implementation of the facade, but not the client-facade interface.	
Clients can use the subclass directly if they need to.	

2.3.3 Behavioral Patterns

Behavioral patterns are used to structure the communication between classes and objects.

Pattern name: Chain of Responsibility

Intent: *"Avoid coupling the sender of a request to its receiver by giving more than one object a change to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it." (Gamma, Helm, Johnson, Vlissides, 1995)*
Applicability: The Chain of Resposibility pattern can be used when more than one object may handle a request in various ways, or if the set of objects that can handle a request should be specified dynamically.
Solution: The structure of this pattern can be seen in figure 2.11. When a client sends a request, it is forwarded along the chain until a ConcreteHandler object responses. If no ConcreteHandler objects can handle the request, the general respons from the Handler class is returned.

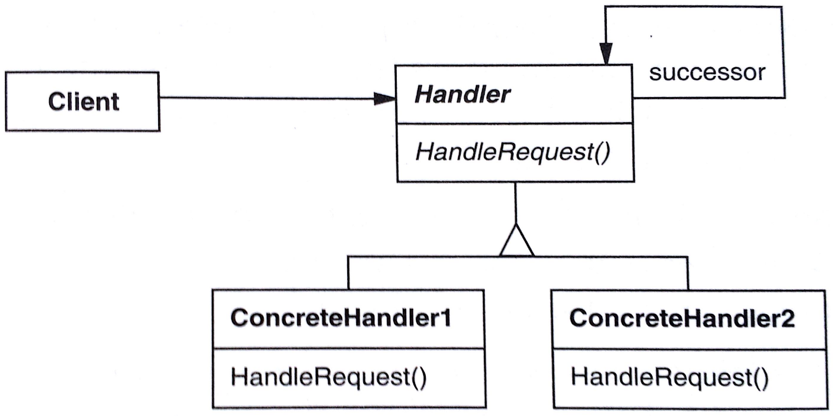


Figure 2.11: Structure of Chain of Responsibility pattern, (Gamma, Helm, Johnson, Vlissides, 1995)[page 225].

Consequences: see table 2.5.

Table 2.5: Consequences of the Chain of Responsibility pattern

Positive consequences	**Negative consequences**
Shields the client from which object actually response to its request.	There is no guarantee the request will be properly handled.
Responsibility can easily be added or removed from the objects.	

Pattern name: Iterator

Intent: *"Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation." (Gamma, Helm, Johnson, Vlissides, 1995)*
Applicability: Use the Iterator pattern to support multiple traversals of aggregate objects, and when aggregate objects and traversal algorithm must vary independently.
Solution: The structure of this pattern can be seen in figure 2.12. The ConcreteIterator keeps track of the traverse, and can be created to traverse in any way, for example as a filter that only returns elements that meets the wanted filtering conditions.

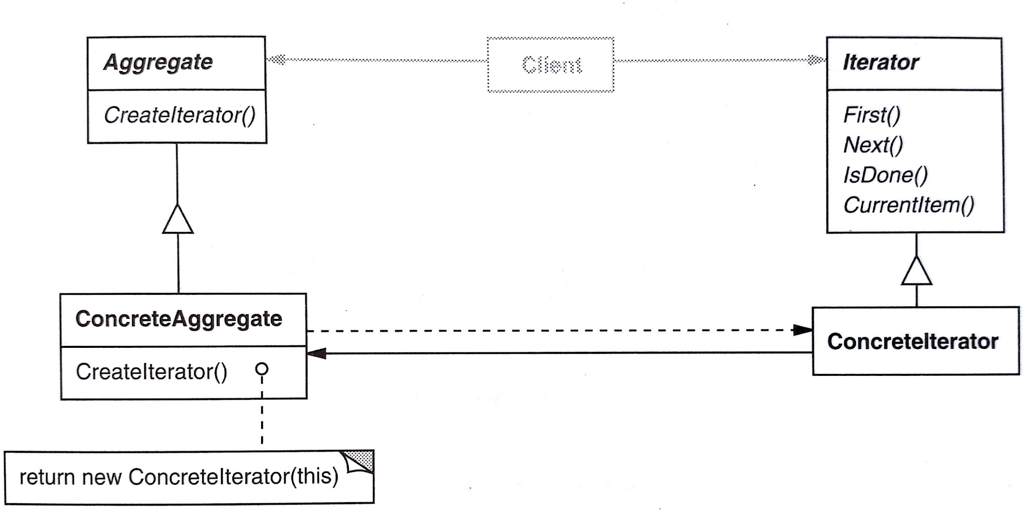


Figure 2.12: Structure of Iterator pattern, (Gamma, Helm, Johnson, Vlissides, 1995)[page 258].

Consequences: see table 2.6.

Table 2.6: Consequences of the Iterator pattern

Positive consequences	**Negative consequences**
Can have variations in the traversal of the aggregate	There can be additional communication between iterator and aggregate object.
The iterators simplify the Aggregator interface	
Multiple iterators can exist for one Aggregator	

Pattern name: Observer

Intent: *"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically." (Gamma, Helm, Johnson, Vlissides, 1995)*
Applicability: Use the Observer pattern when an abstraction has two aspects, one dependent on the other, or when an object should be able to notify other objects without making assumptions about who these objects are.
Solution: The structure of this pattern can be seen in figure 2.13. The Concrete-Subject object notify all its Observer objects when a state is changed for all the objects to be updated.

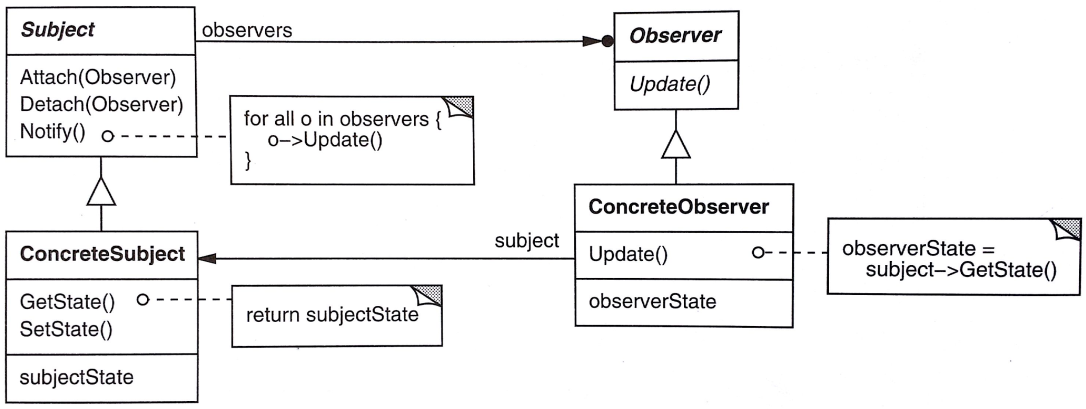


Figure 2.13: Structure of Observer pattern, (Gamma, Helm, Johnson, Vlissides, 1995)[page 245].

Consequences: see table 2.7.

Table 2.7: Consequences of the Observer pattern

Positive consequences	**Negative consequences**
Minimal coupling between Subject and Observer	A large number of observers can cause an overhead in information flow. Small operations may result in a flow of updates to observers and their dependent objects.

Pattern name: State

Intent: *"Allow an object to alter its behavior when its internal state changes. The object will appear to change its class. " (Gamma, Helm, Johnson, Vlissides, 1995)*
Applicability: The State pattern can be used if an object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
Solution: The structure of this pattern can be seen in figure 2.14. The Context object changes State object when its state changes.

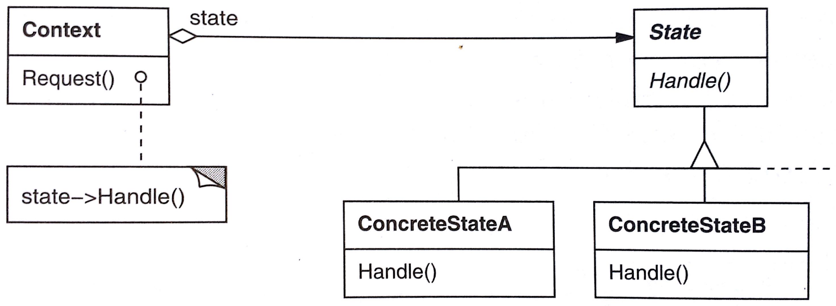


Figure 2.14: Structure of State pattern, (Gamma, Helm, Johnson, Vlissides, 1995)[page 245].

Consequences: see table 2.8.

Table 2.8: Consequences of the State pattern

Positive consequences	**Negative consequences**
State-specific behavior is localized.	Adding a new state requires changes in Complex implementation, that may become complicated.
State transitions becomes explicit.	

Pattern name: Strategy

Intent: *"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it." (Gamma, Helm, Johnson, Vlissides, 1995)*
Applicability: The Strategy pattern can be used if many different classes differ only in their behavior, or if need of different variants of an algorithm. The pattern can also be used if an algorithm uses data that clients should know about.
Solution: The structure of this pattern can be seen in figure 2.15. The clients chooses which ConcreteStrategy to use, but the Context class only knows that the object is a Strategy, and thus it has the same interface with all the subclasses of Strategy.

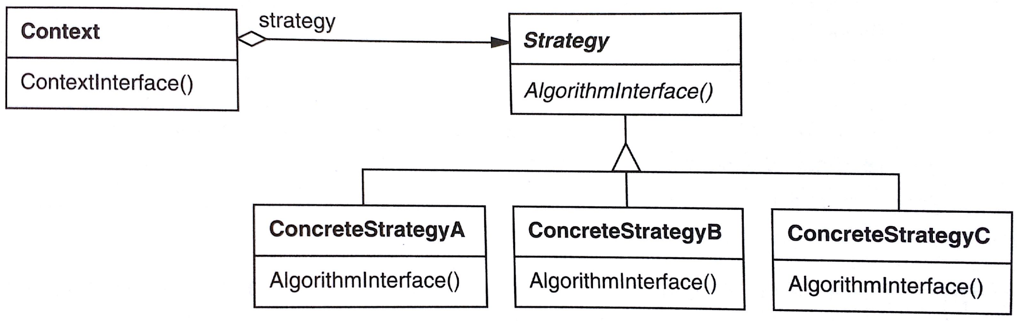


Figure 2.15: Structure of Strategy pattern, (Gamma, Helm, Johnson, Vlissides, 1995)[page 316].

Consequences: see table 2.9.

Table 2.9: Consequences of the Strategy pattern

Positive consequences	**Negative consequences**
Gathers families of related algorithms	Clients must know which Strategy they want to use to optimize the use of the different Strategies.
The Context-Strategy interface remains the same for all Strategy subclasses.	Some ConcreteStrategy classes may not need all the information from the Context class forwarded by the Strategy class, causing a *communication overhead.*
Flexible and reusable code.	Increases the number of objects.

Pattern name: Template Method

Intent: *"Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure." (Gamma, Helm, Johnson, Vlissides, 1995)*
Applicability: The Template Method pattern can be used when the format of an operation is clear, but the aspects of the algorithm vary for each subclass.
Solution: The structure of this pattern can be seen in figure 2.16. The Abstract class implement the TemplateMethod, using the functions implemented in the ConcreteClass.

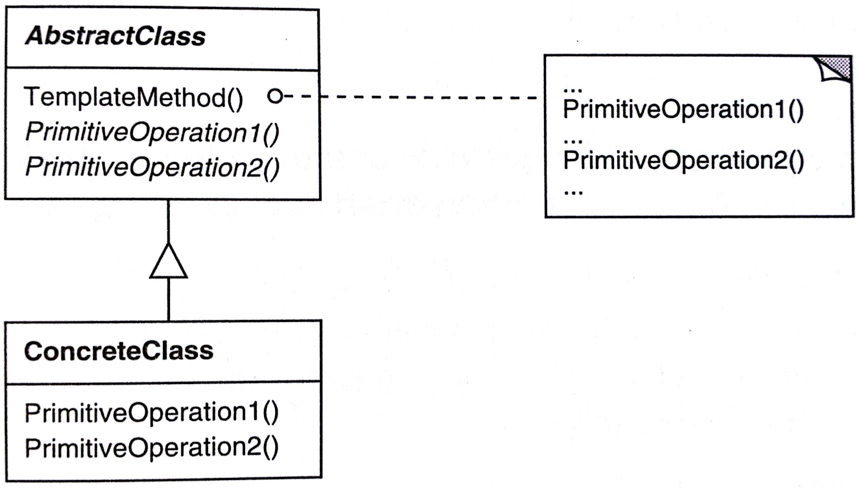


Figure 2.16: Structure of Template Method pattern, (Gamma, Helm, Johnson, Vlissides, 1995)[page 327].

Consequences: see table 2.10.

Table 2.10: Consequences of the Template Method pattern

Positive consequences	**Negative consequences**
Promotes code reuse	Can result in many sub-classes to cover wanted behavior.
Higher probability that sub-classes are implemented with the correct behavior corresponding to the Abstract class	

Chapter 3

Background Theory: Scheduling and Real-Time Systems

3.1 Real-Time Systems

Computing systems have developed from being used exclusively in laboratories and research, to become a part of our everyday lives. Humans interact with computing systems when using telecommunication, multimedia systems, cars, dishwashers etc... These systems are *real-time systems*, i.e. computing systems that react and respond to events in the surrounding environment within precise time constraints. Many definitions exists for real-time systems. A brief but precise definition of real-time systems is:

"Any information processing activity or system which has to respond to externally generated input stimuli within a finite and specified period."
(Young, 1982)

This definition captures the essence of a real-time system, and emphasizes the importance of time. Real-time systems are often classified by the importance of meeting time constraints.
Hard real-time systems have deadlines that cannot be missed, and it is crucial that a response happen within the time limit to avoid total system failure.
Firm real-time systems can miss deadlines occasionally. The system may decay in performance, but will not experience a total system failure. Results processed after deadlines can not be used in any way.
Soft real-time systems can miss deadlines occasionally, and all produced results can be utilized. Missing deadlines causes the system to decay in performance, as the results are less useful after a deadline miss.

(Burns and Wellings, 2009).

The *response time* of a system is the time passed from an input is given until a reaction occurs. The response time of a real-time system is crucial for efficiency, and can create difficult design problems, as one input may induce a longer response time than other inputs. Many real-time systems change their behavior as a response to an occurrence, called an event, i.e. the systems are *event-driven.* The events can be both synchronous and asynchronous, and they influence the resulting output of the system. Events can be initiated by an *interrupt.* An interrupt can origin from both internal and external parts of the system. A system clock can generate interrupts at specific times, creating both cyclic and sporadic interrupts causing synchronous and asynchronous events, (Laplante, 2004).

3.1.1 Parallelism and Processors

A computing system with one single central processing unit is called a *uniprocessor system*, while a computing system with two or more processing units is called a *multiprocessor system* or a *multicore system.* Multiprocessor systems can help increase both efficiency and capacity, but it demands careful management to benefit from the extra processors. It is not a straight forward task to distribute work between processors equally. *Amdahl's law* provides a measurement of how much improvement to expect on an overall system when only one part of the system is improved. Applied to multiprocessor systems, it can be shown that there exists a limit of the number of processors that can be applied to a system, and still be beneficial. At one point, increasing the number of processors can not help speed up the software. Thus, the level of parallelism that can be achieved is determined by the software, not the hardware, (Laplante, 2004). In some scenarios, there is no need for more than one processor. Some processes, such as a calculation, can not be split in two separate parts, and can therefore only run on one processor. In a multiprocessor environment, processors might get stuck waiting for another processor to finish its calculations before proceeding. In a scenario where several processes are dependent on each other, a *parallel* running system will turn into a *serial* execution, making multiple processors redundant. To measure the level of processing, *CPU utilization* can be used, providing the percentage of non-idle processing done by a processor, (Laplante, 2004).

3.1.2 Tasks and Processes

Earlier, programs were written only to execute *sequentially,* but in modern systems, programs are allowed to run more than one thread of execution at a time, and execute them *concurrently*, (Burns and Wellings, 2009). A *task* is a unit of execution,

often used to describe a single thread process. There are variations in the way different programming languages use tasks. The *structure* of a task can be **static**, where the number of tasks are know pre-run-time, or **dynamic**, where tasks can be created at run-time, thus the number of tasks are not know pre-run-time.
The *level of parallelism* describes the hierarchy of tasks in a system. The level of parallelism allowed varies between programming languages. In **nested** systems tasks can initialize other tasks within themselves, as opposed to **flat** systems, where tasks exclusively can be defined in the top level of the program.
The life-time of a task can be separated in several states. A simple state diagram for a task can be seen in figure 3.1. The termination of a task can be caused by a variety of different events. A task may be terminated because it *completes* its execution, it *self-destructs*, a different task *aborts* it, it reaches an *error-condition*, or if there is *no longer need* for the task, (Burns and Wellings, 2009).

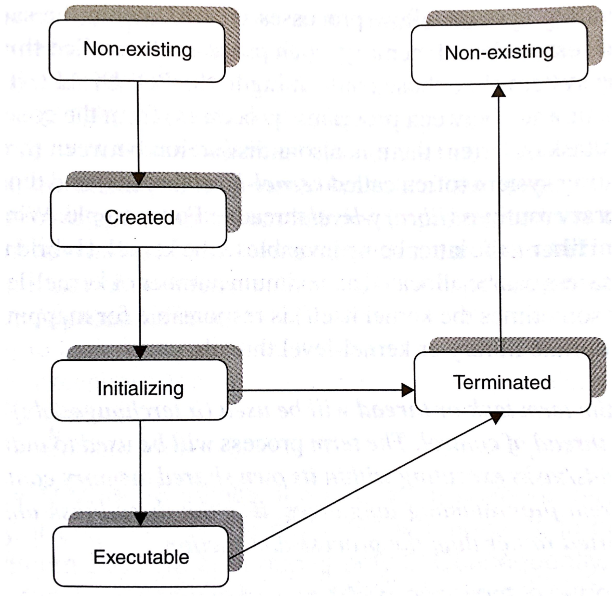


Figure 3.1: A simple state diagram for a task, (Burns and Wellings, 2009)[figure.4.1].

A task is *periodic* if it has a predefined cycle-time. Periodic tasks are time-triggered, as opposed to *event-triggered.* An event-triggered task does not have a specific cycle-time, and is called *aperiodic.* If there also exists a limit on how many times a task may occur within a time-interval, the task is *sporadic.* A set of tasks to be executed sequentially is called a *job.*
The *arrival time* of a task is the time the task becomes available for execution. A *deadline* is a time stamp representing the time when the corresponding task must be completed. An *absolute deadline* is the actual time the task has to be

finished, while the *relative deadline* is the deadline time relative to the arrival time, i.e. the relative deadline is the time from when a task *arrives* until it has to be finished executing. A missed deadline can cause different consequences depending on whether a system is *hard, firm* or *soft*(section 3.1) (Burns and Wellings, 2009).

Concurrent Execution of Tasks

Concurrent execution enables more than one task to execute at approximately the same time on the same processor. This means that the processor can "jump" between different task, as opposed to sequential execution, where one task has to finish in order for another to begin. Concurrent execution can be represented in three basic ways.

- **Fork and join:** The *fork* statement lets the forked process to execute concurrent with the process that invoked the fork. The *join* operation lets the process that invoked the fork and join wait for the procedure currently forked to finish before continuing.
- **Cobegin:** Lets a set of tasks to start concurrent execution at the same time. The end is set to be when the last task in the set finishes.
- **Explicit task declaration:** The task itself specifies whether it will be executed concurrently or not.

(Burns and Wellings, 2009).

In concurrent execution, protecting shared variables is crucial for sustaining correct behavior from the tasks. Unprotected shared variables might result in corrupt data. If task A and task B are sharing a variable x, and task A wants to manipulate it, task A reads the variable first. In the case of concurrent execution, task B might interrupt task A, access x and write a new value to it, before task A interrupts task B and overwrite x based on the old value without considering the changes made by task B. The changes made by B is now disregarded without task B knowing. This can cause unexpected behavior from the system. There are techniques to avoid this from occurring. A *semaphore* is a non-negative integer that can be manipulated with the functions *wait(S)* and *signal(S).* A task that reaches the *wait(S)* function must wait if the value S is greater that zero. If not, it should decrease the value S by one and proceed. When finished handling the shared variable, it calls the *signal(S)* function, which increment the value of S by

one. Another way to protect shared variables is by using a *mutual exclusion mutex.* A mutex works as a lock; if locked, no other task may enter before it is unlocked. A mutex is typically locked when accessing a certain shared element, and unlocked when the task has finished manipulating it. If a different task wishes to access the same element, it must simply wait by the locked mutex until it is unlocked, then enter and lock it itself.
A task can be dependent on a different tasks behavior in order to execute. *Task dependencies* describes the relationships between tasks. There are four basic types of dependencies between a task A and a task B.

- **finish-to-start:** Task A must finish before task B can start.
- **start-to-start:** Task A must start before task B can start.
- **finish-to-finish:** Task A must finish before task B can finish.
- **start-to-finish:** Task A must start before task B can finish.

(Mall, 2007)
These dependencies do not insist on task B to start/finish the moment it *can*, but at any point *after* task A has started/finished.
Both shared-variable protection and dependencies can induce a *deadlock.* A deadlock is a situation where no tasks can execute because they are all waiting on a variable currently locked or their dependency task to finish/start. A common deadlock situation is when task A and task B both needs the same two variables x and y. Task A locks variable x and task B locks variable y, and now both tasks are waiting to lock the other variable, while continuing to keep the first variable locked. Situations like this causes a system blockage, and must be avoided. A possible backup solution is to make one of the tasks release their first variable, or by using a method called *atomic actions.* Atomic actions allows an operation to appear invisible to the rest of the system. Task A can use atomic actions to see if it can lock both variables, and if it can, the action happens instantly, and if not, no action will be taken, (Burns and Wellings, 2009).

Task Execution Time

Worst-case Execution Time, denoted WCET, is, as described in Burns and Wellings (2009), a measure for the longest possible time it will take for a task to complete its body of execution. An estimation of the WCET is important information, that can help the overall productivity and efficiency of a system. There are several ways to obtain the WCET value, mostly involving measuring and/or analysis. Measuring it can lead to the question "was this the absolute worst-case time?". Analyzing the WCET requires a good model of the processor, which is not always available. The *average execution time* is an estimation of the average time it would take for a task to finish, while the *burst time* is the actual time it will take a task to execute.

3.1.3 Scheduling

When working with real-time constraints, distributing the available processing time becomes an important part of creating a successful system. To share processing time in a logical and optimal fashion, *scheduling* can be crucial. The *scheduler* decides which task the processor should execute from a set of tasks ready to be executed. The logic behind this decision depends on the schedulers algorithm, often based on parameters like worst-case execution time, deadlines, periods, etc... Some schedulers logic favors some tasks over others, risking some tasks to be left unexecuted, i.e. some task may suffer *starvation.* There are many ways of classifying different types of schedulers. One of them is based on the level of interruption.

A *co-operative* or *non-preemptive scheduler* is a scheduler that lets the processor finish a task before scheduling a new task to the processor.

A *preemptive scheduler* can interrupt the processor while it is executing a task, and force the processor to start executing a different task instead. This switch from one task to another is called a *context switch*, and it causes some overhead time loss.

Not all processors can be interrupted and can thus only use non-preemptive schedulers. (Burns and Wellings, 2009)

Response Time Analysis

Response time is, as mentioned in section 3.1, important for a real-time system. Thus, using the estimated computation time for each task, the response time can be estimated, using *response time analysis.* If the response time estimated for a specific task exceeds the task's corresponding relative deadline, the task is not *schedulable.* There are different ways to estimate the response-time of a task given a specific scheduler, often involving an estimation of the execution time (WCET or average execution time), priorities set by the scheduler, and the level of interference with other tasks. There are some rules connected to the outcome of the response-time analysis.

- Using average execution-time estimates and average arrival-time estimates, all task should be schedulable.
- Using WCET and worst-case arrival times, all *hard real-time* tasks should be schedulable. (See section 3.1 for definition of "hard real-time system").

(Burns and Wellings, 2009)

Using response time analysis, a scheduler can be tested to see if a task set is schedulable or not. Using the average execution-times, a task set may be schedu-able but still miss deadlines during the actual execution. These situations are called a transient overload, as it is a temporarily overload of the system. If a system is scheduable using the WCET estimations, a transient overload should not occur. Closely related to response time is *turnaround time*. This is the time from when a task is submitted to the system until the task has completed its body of execution. High turnaround time indicate loss of deadlines and task starvation.
(Burns and Wellings, 2009)

Online Analysis

There exist hard- and soft-real-time systems, and schedulers must accommodate the possible different needs the different systems may have. Not all systems have the possibility to estimate WCET and arrival-time patterns of their tasks pre-runtime, thus *online analysis* is needed. An online analysis scheme limits the number of tasks that can compete for the processor, and uses a scheduler for these tasks to control the task execution submissions. To choose which tasks to allow competing for the processor, a value based system is often used, assigning each task a value based on its properties. The value-system is said to be *static* if the tasks value can be calculated regardless of its release-time, *dynamic* if the tasks value can be calculated only at the time of release, *dynamic* if the tasks value can change during execution, (Burns and Wellings, 2009).

Priorities

A task can be given a priority by the scheduler, based on the nature of the task. The Priority assignment can happen in different ways.

- **Static:** The tasks are assigned priorities pre-run time.
- **Dynamic:** The tasks are assigned priorities at run time.
- **Adaptive:** The tasks are assigned priorities which may change during run time.

When a task is ready to execute, but is currently unable because a higher priority task is being executed, it suffers *interference*. If a task that is ready to execute continuously suffers interference, it may cause the task to *starve*, missing its deadline, and never being executed. *Priority inheritance* is a term used to describe a solution to a situation where a high priority task is blocked from executing because it is

waiting for a lower prioritized task to either finish or unlock a certain variable. *Priority inheritance* lets the lower prioritized task *inherit* the higher priority from the other task temporarily, until the resources necessary for the highest priority task to continue are released. If a task is waiting to execute in favor of a lower prioritized task due to priority inheritance, the task is said to be *blocked.*
(Burns and Wellings, 2009).

Existing Scheduling Disciplines

There exists a large number of scheduling disciplines. Some are more common and known than others, such as the ones listed below.

First **I**n, **F**irst **O**ut

Commonly known as FIFO, or FCFS*(First Come, First Served*). A simple non-preemptive scheduler, that simply lets the processor handle the task with the earliest arrival time, i.e. the first task to arrive at the system gets handled first.(Shaw, 2001)

Round **R**obin

Round Robin, RR, is a scheduling policy without priorities, that assigns a fixed time interval of execution to each task, and cycles through them. RR is preemptive, as it may have to interrupt a running task at the end of the time interval in order to execute the next task. (Shaw, 2001)

Rate **M**onotonic

Rate Monotonic, RM, is a static priority scheduler, assigning the tasks priorities based on their cycle duration. Shorter cycle gives a higher priority. RM is preemptive, beneficial for the highest prioritized task.(Burns and Wellings, 2009)

Earliest **D**eadline **F**irst

Earliest deadline first, EDF, is a dynamic scheduling algorithm, assigning tasks priorities relative to their absolute deadline. In run-time, EDF will search the task set for the task closest to its deadline. This task will then be executed by the processor. (Burns and Wellings, 2009)

Shortest **J**ob **F**irst

Shortest Job First, SJF, is a non-preemptive scheduling algorithm, that selects the job with the shortest estimated execution time to execute first. If new jobs with short execution times are frequently admitted to the system, other tasks with longer execution times may suffer starvation. (Shaw, 2001).

Gantt Chart

To illustrate the executions performed by the processor, a *Gantt chart* is commonly used. A Gantt chart is a time dependent bar chart, with tasks or activities listed along the vertical axis, and time along the horizontal axis. Normally, for real-time scheduling, Gantt charts can illustrate the different tasks executing, waiting, locking resources, etc. A Gantt chart gives a general idea of the resulting CPU behavior a scheduling policy will cause. An example of a Gantt chart marking execution of three tasks, using the Round Robin scheduling policy can be seen in figure 3.2, (Burns and Wellings, 2009).

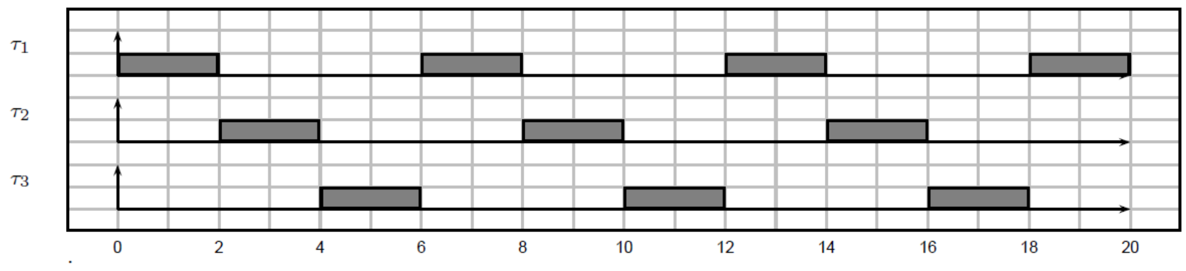


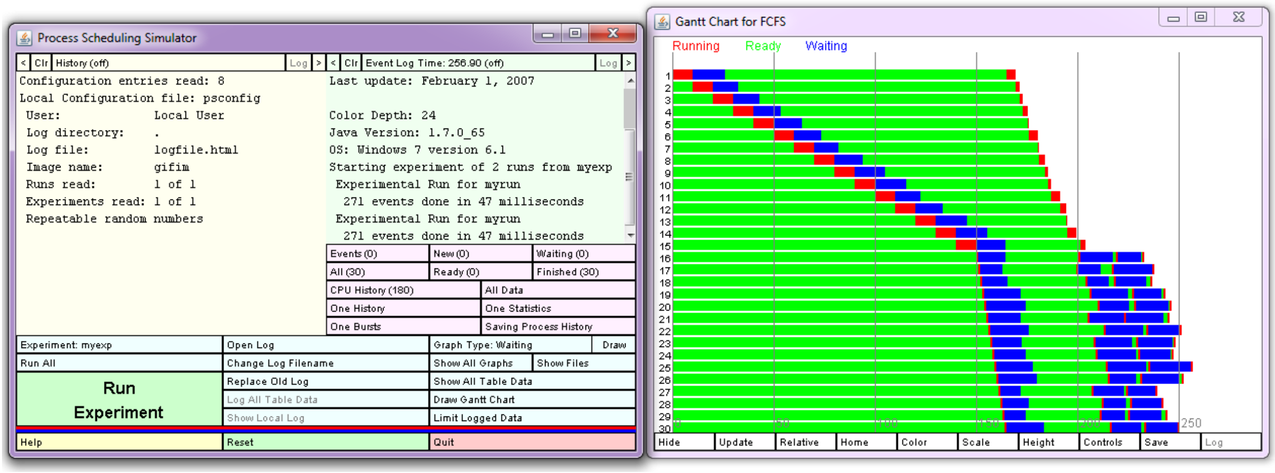
Figure 3.2: Gantt chart showing processor execution in grey. Three tasks using a Round Robin scheduler.

3.2 Existing Scheduling Simulators

There exist several simulators for simulating processor behavior given a certain scheduling policy, similar to the software to be developed in this project. Some are text-based, and some have a Graphic User Interface (GUI).

3.2.1 Process Scheduling Simulator

The Process Scheduling Simulator, PS, is a Java-application, created to let the user experiment with different scheduling strategies through a GUI, (Robbins, 2007). Processes to be simulated consist of four values: Arrival time, total cpu time needed, cpu burst time distribution and I/O burst time distribution. The burst times are described by probability distributions. After simulating the processes, Gantt charts can be generated and examined. A screenshot of the application can be seen in figure 3.3.



```
Configuration entries read: 8
Local Configuration file: psconfig
 User:               Local User
 Log directory:      .
 Log file:           logfile.html
 Image name:         gifim
 Runs read:          1 of 1
 Experiments read: 1 of 1
 Repeatable random numbers
```

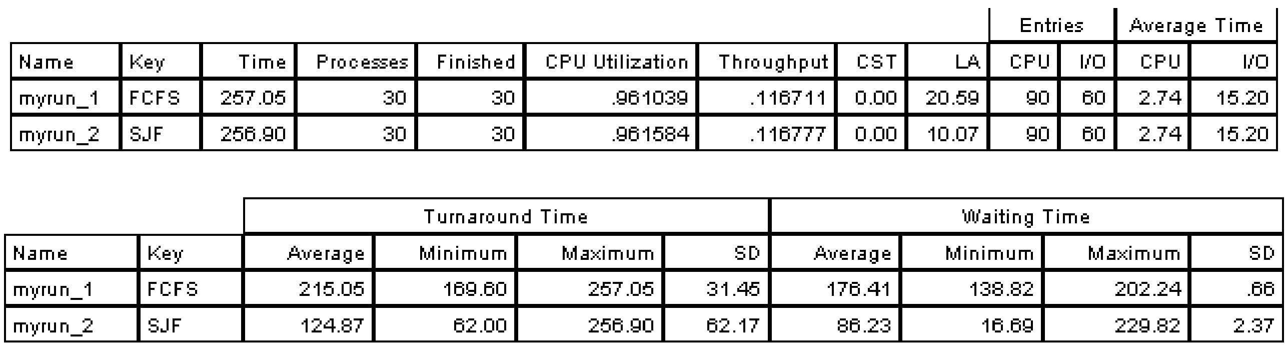
```
Last update: February 1, 2007

Color Depth: 24
Java Version: 1.7.0_65
OS: Windows 7 version 6.1
Starting experiment of 2 runs from myexp
 Experimental Run for myrun
  271 events done in 47 milliseconds
 Experimental Run for myrun
  271 events done in 47 milliseconds
```

Figure 3.3: A screenshot of the Process Scheduling Simulator application, with resulting gantt chart.

Supported scheduling policies are listed below.

- **F**irst **I**n, **F**irst **O**ut
- **S**hortest **J**ob **F**irst
- **R**ound **R**obin

After running an experiment, a table of related statistics and calculations can be generated for the simulated schedulers, see figure 3.4.



									Entries		Average Time	
Name	Key	Time	Processes	Finished	CPU Utilization	Throughput	CST	LA	CPU	I/O	CPU	I/O
myrun_1	FCFS	257.05	30	30	.961039	.116711	0.00	20.59	90	60	2.74	15.20
myrun_2	SJF	256.90	30	30	.961584	.116777	0.00	10.07	90	60	2.74	15.20

		Turnaround Time				Waiting Time			
Name	Key	Average	Minimum	Maximum	SD	Average	Minimum	Maximum	SD
myrun_1	FCFS	215.05	169.60	257.05	31.45	176.41	138.82	202.24	.66
myrun_2	SJF	124.87	62.00	256.90	62.17	86.23	16.69	229.82	2.37

Figure 3.4: A screenshot of a table of resulting data after a simulation in the Process Scheduling Simulator application

3.2.2 CPU Scheduling Simulator

CPU Scheduling Simulator, Cpuss, is a simulator written in C# 3.0, created by Granville Barnett in 2008, (Barnett, 2008). It is meant to let the user gather metrics on scheduling algorithms, and create and implement new algorithms. Implementing a new scheduling algorithm requires a implementation of the *IStrategy* interface, which contains two methods:

- `void Execute(Runner runner);` Contains the algorithm of the scheduler.
- `string ToString();` Returns a string that includes the name of the scheduler.

A Runner is a type that takes two arguments: a processLoad, which is a set of processes, and a scheduler algorithm. The Runner is then used to invoke the simulation by calling the method: Runner::Run(). The simulation offers four events,: *ProcessStarted, ProcessPreempted, ProcessResumed and ProcessCompleted.* They can be used to print out events to follow the progress in the simulation.
Cpuss has a report generator, which allows a user to perform a test scenario using a specified scheduling strategy. Test processes are autogenerated in three different sizes: small, medium and large. The report consists of graphs and matrices on relevant information, like response-time. The `Execute` method in the scheduling strategy is supposed to take a runner type object, put the processes in the correct order, and tell the CPU to execute them by using the runner function: `UtilizeCpu(process)`. This means that the scheduler only has to run its `Execution` algorithm one time the a runner object. The user thus needs to be familiar with the runner type to successfully implement a new scheduling algorithm. Pre-implemented schedulers are listed below.

- **F**irst **I**n, **F**irst **O**ut
- **S**hortest **J**ob **F**irst
- **R**ound **R**obin
- **P**riority **F**irst

3.2.3 SCHEDuler SIMULAtor

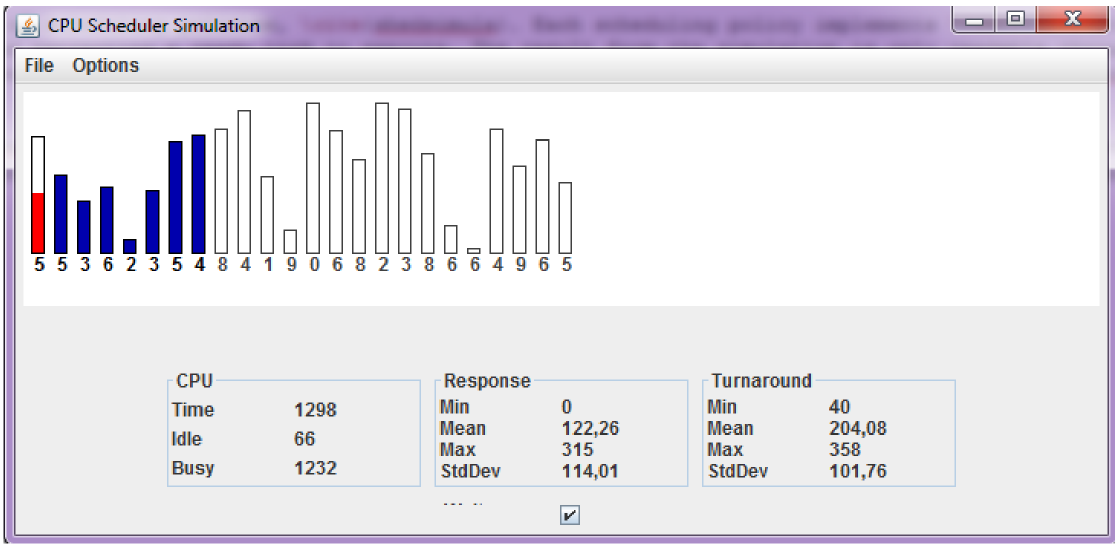
SchedSimula is an event driven C++ simulator for a set of pre-implemented scheduling policies (University of Padova, 2012). Each scheduling policy implements the functions `void insert(Task p)`, which are methods for gives the scheduler a ready task, and `Task extract()`, which returns a ready task to execute. The result from the simulation is only readable through KIWI Format Viewer. The simulator can simulate periodic task sets, and five scheduling policies, which are listed below.

- **F**irst **I**n, **F**irst **O**ut

- **L**ast **I**n, **F**irst **O**ut
- **E**arliest **D**eadline **F**irst
- **R**ate **M**onotonic
- **D**eadline **M**onotonic

3.2.4 CPU Scheduler Application

The CPU Scheduler Application is a Simulator written in Java2 SDK, (Weller, 2006). It is an application with a Graphical User Interface,GUI, created to simulate a randomly generated set of 50 processes with different schedulers. The application simulates the progress of the processes, and lets the user change the scheduling algorithm during the simulation, see figure 3.5.



CPU	
Time	1298
Idle	66
Busy	1232

Response	
Min	0
Mean	122,26
Max	315
StdDev	114,01

Turnaround	
Min	40
Mean	204,08
Max	358
StdDev	101,76

Figure 3.5: CPU Scheduler Application. Current algorithm being used: FIFO. Ready processes are marked in blue. Process currently executing is marked in red.

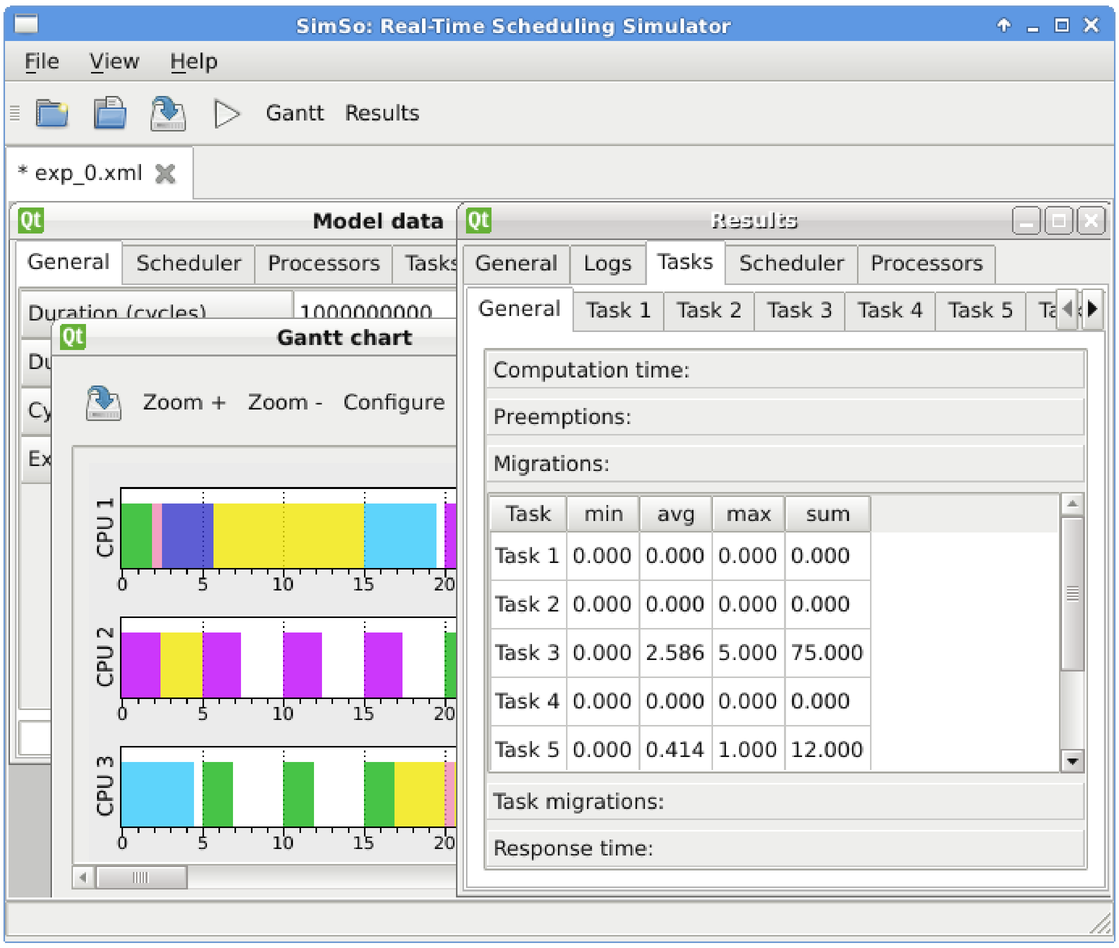
Information about response- and turnaround time is being updated in real-time, and the progression of the simulation can be paused and unpaused. The auto-generated process-set can be saved and reloaded to compare scheduling algorithms. The processes have three parameters: **initial burst time**(execution time), **delay**(arrival time), and a **priority** from 0 to 9. There are four supported scheduling algorithms in this application.

- **F**irst **I**n, **F**irst **O**ut

- **S**hortest **J**ob **F**irst
- **R**ound **R**obin
- **P**riority **W**eighted *(using randomly set priorities)*

3.2.5 SimSo

SimSo is s scheduling simulator based on the python written discrete-event simulator SimPy 2.3.1, (Chéramy, 2014). This simulator centers around multiprocessor architectures, with the ultimate goal to simulate modern schedulers in a multiprocessors-system with all overheads included. The current SimSo simulator can simulate the loss due to context switching and scheduling decisions as overheads. Through a GUI interface, the user can generate task sets, choose scheduling algorithms, set up processors, and run the simulation, see figure 3.6.



Computation time:

Preemptions:

Migrations:

Task	min	avg	max	sum
Task 1	0.000	0.000	0.000	0.000
Task 2	0.000	0.000	0.000	0.000
Task 3	0.000	2.586	5.000	75.000
Task 4	0.000	0.000	0.000	0.000
Task 5	0.000	0.414	1.000	12.000

Task migrations:

Response time:

Figure 3.6: Screenshot of the SimSo GUI, image from (Chéramy, 2014).

Simso can also be used as a library, using the scrips and source code instead of the GUI. The source code consists of four main modules.

- **`simso.core`**: consists of classes needed for simulating.
  - `Scheduler`
  - `Task`
  - `Job`
  - `Model`
  - `Processor`
  - `Timer`
  - `Logger`
- **`simso.configuration`** contains a simulator configuration.
  - `Configuration`
- **`simso.generator`** Generates tasks.
- **`simso.utils`** Used for multiprocessor simulations.
  - `PartitionedScheduler`

SimSo currently supports 25 schedulers, the majority customized for multiprocessor systems. The uniprocessor schedulers are listed below.

- **E**arliest **D**eadline **F**irst
- **R**ate **M**onotonic
- **F**ixed **P**riority
- Static-**EDF**
- **CC-EDF** (Real-time dynamic voltage scaling)

The multiprocessor schedulers implemented in SimSo can be classified within the following categories:

- Uniprocessor schedulers adapted to multiprocessor
- Partitionned
- PFari

- DPFair
- Semi-Partitionned

New scheduling policies may be added to SimSo. By creating a new class that inherits from the `Simso.core.Scheduler` class, a set of methods must be implemented. The methods are listed below.

- `def init (self):` Called when the simulation is ready to start. Initializing structures used by the scheduler must happen here, and not in the constructor.
- `def on_ activate(self, job):` Called when jobs are activated.
- `def on_ terminate(self, job):` Called when a job finishes its execution.
- `def schedule(self, cpu):` The function called by the processor when the scheduler needs to run.

When all these methods are implemented in a new class, the name of the class must be the same as the filename. The new scheduling policy is implemented.
To run a simulation, a `configuration` must be created. The `configuration` is a composition of tasks, duration, processors and a scheduler. The `configuration` is then added to a `model`. The simulation is started using the method `"simso.run_ model()"`.
Task sets to be simulated can be autogenerated. Currently, both periodic and sporadic tasks are supported, but to create sporadic tasks, activation dates must be manually specified.
The simulation results provides an overview of the *overhead and system loads* that were induced by the simulation, a **log**, stating when tasks started and ended. A **Gantt** chart can be generated after the simulation, showing how the CPU/CPUs executed the different tasks in the task set.

3.3 Software and Tools

To conduct the software development project for this thesis, a set of existing open source tools were used.

3.3.1 yEd Graph Editor

yEd is a graph editor, used to create, arrange, import and export diagrams and graphs. This editor was used to create class diagrams, use-cases, flow charts, state machine charts, and in general diagrams to illustrate system behavior and structure. yEd has an intuitive user interface, based mainly on a *drag-and-drop* technique. (yWorks, 2015)

3.3.2 Microsoft Visual Studio

Microsoft Visual Studio is a development environment for Windows, with included compilers for i.a. C/C++. To sustain cross-platform properties for the software, no specific Microsoft libraries were used during the development. (Microsoft, 2015)

3.3.3 rtsched Latex Style

rtsched is a LaTeXstyle created by Giuseppe Lipari to generate Gantt charts. Through a set of commands, a diagram is created, and tasks can be graphically presented as arrived, executed and interrupted.
(Lipari, 2005).

Chapter 4

Work and Development

4.1 Evaluating Existing Solutions

When creating a new software, evaluating already existing solutions can be helpful to map useful features. Based on the solutions described in section 3.2, an evaluation was conducted, first based on each solution, and finally by reviewing the features demonstrated by these solutions.

4.1.1 Process Scheduling Simulator

The Process Scheduling Simulator has a GUI to execute simulations, which gives the user a visual input when setting up the simulation. The GUI in this case is very complicated with a large amount of buttons. This may not simplify the simulation set-up, but work against its purpose and confuse and discourage the user. A Gantt chart can be generated after running the simulation, giving a visual representation of the process-execution time line. The Gantt chart from the PS, shown in figure 3.3 has an untraditional Gantt chart layout, but it is still serving the purpose of a visual presentation of the processors work. The processes used in the simulation does not have a deadline variable, and thus, deadline-misses can not be recorded. This simulator can be a suitable tool to get a visualization of how the FIFO scheduler behaves compared to the SJF scheduler, as it also provides a statistic table of the two schedulers performances, shown in figure 3.4. The simulator falls short of actually comparing the two algorithms through measuring missed deadlines, which is an important factor for a real-time system. There is no option to add new schedulers to the system, and it is not tailored to manually specify new task sets.

4.1.2 CPU Scheduling Simulator

The CPU Scheduling Simulator is a text-based simulator, offering the user the possibility to add new scheduling policies. The new scheduling policies has to implement two methods, with the `void execute(Runner runner)` being the more significant of the two. This method has to call on the CPU-simulator, using the runner object, to execute processes. This demands that the user has great knowledge of the Runner-type and the overall flow of the simulation. Thus the `execute` algorithm can be demanding to create, and puts a lot of responsibility on the user. Processes can be auto-generated in three sizes: small, medium and large, which simplifies the process-initialization. The processes have no deadline, and thus deadline-misses can not be logged.

4.1.3 SCHEDuling SIMULAtor

This text-based simulator written in C++ is obviously in the beginning of development, judging by the little available documentation and the limited user-friendliness. Even though no user tutorial or description exists, it could be possible to add new scheduling algorithms, following the pattern of the other already implemented scheduling policies. This means that the user has to implement a comparing class to sort out the highest prioritized processes, and understand the correct behavior of the simulated processor. This can be challenging. The resulting output from the simulation can only be read through a software called KIWI. The simulator only supports periodic tasks, and they do not have deadline variables, and thus, deadline-misses can not be logged. All in all, this is a simulator with limited flexibility, which requires a user who understands the bits and pieces of the entire software to utilize it properly.

4.1.4 CPU Scheduling Application

The GUI for this scheduler is visual and intuitive to understand and use. The play/pause function lets the user easily manipulate the time and examine specific areas. The real-time response from the system, showing process execution progress, provides a simplistic way to understand how the pre-implemented schedulers work. The processes are auto-generated and can be saved if the user wishes to reuse the same set of processes in a different simulation, but the processes themselves can not be specified through the GUI, nor do they have deadline-variables. The user is not meant to be able to add new scheduling policies for this simulator, and so, only the pre-implemented scheduling policies can be used in simulations.

4.1.5 SimSo

The python based SimSo provides the user with freedom to add scheduling policies, generate and customize task sets and processors through the GUI. The simulation results provides Gantt charts and logs of tasks start time, abort-time and completion-time. The deadline misses are poorly marked in the Gantt charts, and are only shown in the tasks-logs if the specific task is marked for: *"abort in case of deadline miss"* pre-simulation. Even then, spotting the aborted task in the log can be difficult. Many of the features in SimSo are created to fit multiprocessor systems, and they become redundant for uniprocessor systems. The possibility to create new scheduling policies demand the user to implement four algorithms within strict constraints for the simulation to work. The scheduler also has to create and administer its own task-container. Knowledge of python is acquired.

4.1.6 Evaluation of Features

Based on the evaluation of existing scheduling simulators, a list of features to avoid and to preserve in the new software emerged.

GUI can be useful to create a platform for the user to create simulations, but if overly complicated the user may find it confusing. If the user has the possibility to create new scheduling policies, they may have to use an editor and familiarize him or her self to the corresponding programming language and system architecture any way, and a GUI can be redundant.

Creating new scheduling policies is a great way for the user to utilize the simulator in a customized way, but the implementation should not require too complicated behavior from the scheduler, nor force the user to get overly familiarized with the rest of the system.

Creating tasks to simulate is easy when they are auto-generated, and the possibility to save a task-set to rerun it with a different scheduler facilitates comparing two or more scheduling policies. A task should have a corresponding deadline to be used to log any deadline misses. Specifying variables for the task, like size, deadline, etc can be useful when using schedulers that favors some task properties over others.

Results and logs are useful to record the behavior of the simulated scheduler. Gantt charts provide a visual presentation of the processors behavior, and deadline misses should be clearly marked. Deadline misses affect the quality of a real-time system, specially *hard* real-time systems, see section 3.1. Other useful statistics should be included. The report containing the results and logs should have a common, easy-to-use format.

4.2 Software Development Project: Scheduling Simulator

For the software development project for this assignment, the supervisor of the thesis operated as the *client*, while the author operated as the *developer*. The software was created using a mix of strategies from various software development methods. The simulator was created and tested through incremental iterations. The documentation for this software can be found in chapter 6. The software was created in C++ programming language. This language is commonly used for real-time systems, and since this software is meant to facilitate choices regarding scheduling for developers of real-time systems, using a language a potential user is familiar with is nothing but logical. C++ also provides the possibility to easily use design patterns, described in section 2.3, as it is object oriented.

4.2.1 Choosing Software Development Method

The new scheduling simulator software, created by the author, was developed based on the best practices within software development described in section 2.1. As mentioned, a common practice is to "pick and mix" strategies from different methods to fit the nature of the relevant project. The final "picked-and-mixed" list of strategies and methods are listed in table 4.1.
In this project, as the developer is learning theory and developing at the same time, a strict step-by-step model, with little flexibility, like the **Waterfall model** described in section 2.1.1, seemed like a non-suitable software development method for this case. **Incremental and iterative methods**, described in section 2.1.2, on the other hand, are more flexible, allowing the developer to design, create and learn from the process and try again. **Agile development methodologies**, described in section 2.1.7, utilizes iterative and incremental strategies, and are often used in today's developing projects, as the Agile methods are light, open-minded and emphasize a social work space. The project in hand was to be developed by one person, and so the factor of sociability, such as "pair-programming" in extreme programming, section 2.1.9, has no significance for the developer and her work. The **Scrum** development method is primarily created for a team of people, see section 2.1.8. The social aspects are also here essential and mostly irrelevant for this project, but *the daily scrum* could still be useful, as it promotes self-reflection and encourages all progression to be in a goal-orientated direction. *The sprints* are also applicable to this project, as they help dividing a large goal into smaller, more achievable goals. Frequent contact with the costumer is mentioned to be essential throughout all the light-weight methods, be it though prototyping, early releases or just through meetings and written reports. The contact is there to keep

the developer from creating a product that does not meet the expectations of the costumer and to keep the developers focus on the requirements. Another way to preserve requirements is simplicity, supported by the software philosophy to **K**eep **I**t **S**imple, **S**tupid.

Table 4.1: "Picked-and-mixed" Software Development Methods

`Method`	**Which qualities and why**
`Scrum`	The daily scrum, and sprints
`Iterative and Incremental Development`	Related to sprints. Increasingly larger software through cycles of development.
`Agile Development`	Light and flexible development. Emphasizing working code.
`KISS`	Simple behavior to explain may perhaps lead to good code.

4.2.2 Specifying Requirements

The supervisor of this project had a vision of what properties the software should have. The supervisor thus took the part of a "*client*", while the student working on this project was the "*developer*". To understand the functionality the client wanted from the software, a client-developer meeting was held. In section 2.2.1 it is described that when communicating with a client, and when creating a specification, general language is must be used. A set of questions were created, using common words rather than technical terms. By asking open ended question, the developer invoked the client to fill in with thoughts and wishes, facilitating the uncovering of what the client really wanted, rather than filling in technicalities based on the developers preset understanding of what the final product should be. The pre-made questions for the meeting were:

- What kind of software tool is this going to be?
- What is the wanted result?
- What is the input and output of this software?
- Who is the user of this software tool?
- How is the software tool to be used?

- What separates this simulation tool from other existing tools?
- What functionality should the software tool have?

The actual client-developer meeting had a dynamic dialog, and so the questions were altered to fit the conversation and to benefit from the topics discussed. A report from the meeting can be seen in A.1. The software requirements were specified using the information from this meeting, the assignment text for the project, and the evaluations of other existing solutions in section 4.1. Based on the requirements for the new software, the key features from the evaluation of the existing solutions were reviewed again, as shown below.

GUI is not necessary. A text based user interface is sufficient, as the user of the software is intended to be a person with knowledge in C++

Creating new scheduling policies shall be possible in the new software, and the simulator shall be as open as possible for new scheduling policies to be added and simulated.

Creating tasks to simulate shall be possible in the new software, both by creating them manually and by generating a task set. Task shall contain enough information to record potential deadline misses.

Results and logs shall provide any useful statistics and information, both graphical and numerical. Gantt charts are preferably a part of the output.

Representing the requirements in different formats is a way to preserve the wanted functionality later in the design process. Use-cases were used to separate and identify the different levels of interaction between the user and the simulator, and the interaction between the processor simulator and the scheduler. Since the developer used mainly an iterative Agile software development method, the requirements were not fixed at the beginning of development, but they evolved through conversations with the client. The Requirement Documentation can be seen in section 6.1.

The *unit of time* used in the simulator software is irrelevant, as long as all time elements are set proportional to each other. For the sake of clarity, the time unit `ms` will be used though the development and software description, but the mention of a time unit will have no impact on the resulting output from the simulator.

4.2.3 Iterative and Incremental System Development Process

The Agile development methodologies suggests that incremental and iterative improvements are desirable when developing a program. *Sprints*, described in section 2.1.8, are a form of iteration cycles used in Scrum. Goals are set, and are attempted to be completed by the end of the sprints duration. This software was developed using incremental sprint-like iterations, where the iterations are restricted by requirement goals, and not by duration. A basic simulator was developed first, and gradually improved and expanded to fit the mapped requirements. The following sections describes the iterative evolution of the software and design choices. Further class-descriptions and class-design choices are described in section 4.2.4. Four iterations were completed to reach a level where the software adequately satisfied the requirements.

First Iteration: Core Functionality

In the first iteration of development, the basis of the wanted software behavior was created. The core functionality of the software is specified in the following requirement.

Requirements:

- The software simulates a scheduler and a task set.

In order to simulate a scheduler and a task set, two objects are required in addition to a platform where the simulation can take place. The scheduler needs to be able to schedule a task on demand, and a task should have some properties the scheduler can base its choice on. A task set demands some sort of a container for task objects. Based on this, a simple class diagram was created, shown in figure 4.1.

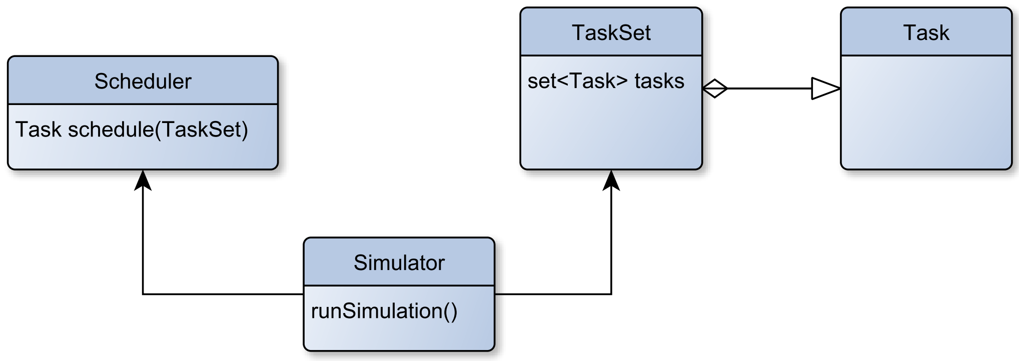


Figure 4.1: Basic system class diagram, first iteration

.

`The Scheduler class` is a class representing a scheduling policy. The function "`Task schedule(TaskSet)`" gives the scheduler a taskset, lets the scheduler access all the tasks, and returns the task best suited for execution, according to the scheduling logic.

`The Task class` is a class representing a task, 3.1.2.

`The TaskSet class` is a container for `Task objects`.

`The Simuator class` is a class where the scheduler and the task set is combined and the simulation is conducted.

What was learned from this iteration: The simulator must contain a scheduler a task set.

Second Iteration: Simulating a RM Scheduler on a Specified Task Set

Incremental developing expands the software portion of interest for each iteration. The second iteration included a new requirement of interest.

Requirements:

- The software simulates a scheduler and a task set.
- The output of the system consists of relevant plots and graphic representations of the results.

The goal for this iteration was to simulate a Rate Monotonic Scheduler for a specific task set, gather relevant information, and generate an output consisting of plots and graphic representations of the results. A simplified class diagram for this system can be seen in figure 4.2. The output produced by this early implementation was a simple Gantt chart, see section 3.1.3, created with ASCII characters in a .txt document. Being able to access all tasks enabled the RM scheduler to assign priorities to the tasks based on their period before the simulation. The `Model` class created the output file with a Gantt chart based directly on the preset task priorities.

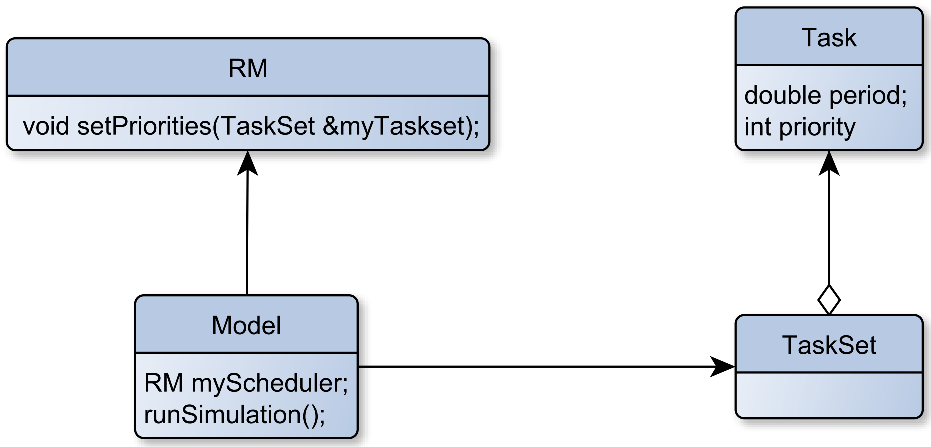


Figure 4.2: Simplified system class diagram, second iteration

.

`The RM class` is a class representing the Rate Monotonic scheduling policy. It assigns priorities to all the tasks in the task set, based on their period-time.

`The Task class` is a class representing a task.

`The TaskSet class` is a container for `Task objects`.

`The Model class` is a class where it calls upon the RM scheduler to set the tasks priorities, and then uses the task priorities to create a ASCII-based Gantt chart.

The complete class diagram can be seen in A.2.1. A resulting output from this system can be seen in A.2.2.

What was learned from this iteration: The software developed in this iteration could only simulate periodic tasks using an RM scheduler. The final product must be able to simulate several schedulers, thus the connection between the model class and the schedulers class must be less dependent on the schedulers type. Also, the simulators behavior should not depend on task priorities, as it does in this system.

Third Iteration: Simulator for all Schedulers with Specified Task Sets

Requirements:

- The software simulates a scheduler and a task set.
- The output of the system consists of relevant plots and graphic representations of the results.
- The simulator is openly created to simulate new scheduling types.

The second iteration system included only one type of scheduler to simulate. In this iteration, a system allowing new schedulers to be implemented and simulated was created. The design pattern Strategy, from section 2.3, is applicable. The schedulers can be seen as a *"family of algorithms"*, and the Strategy pattern lets the scheduling *"algorithm vary independently"*. Combining this with the Template Method pattern, where *"the skeleton of an algorithm"* is defined, the *"the subclasses redefine certain steps if the algorithm ..."*. Based on figure 2.15 and figure 2.16, the structure for schedulers in figure 4.3 was created. This design allows new scheduling policies to inherit the scheduler class, and define their own logic in the scheduling algorithm. The external system can only see a Scheduler class, and can utilize the same interface regardless of which Scheduling sub-class the system is interacting with.

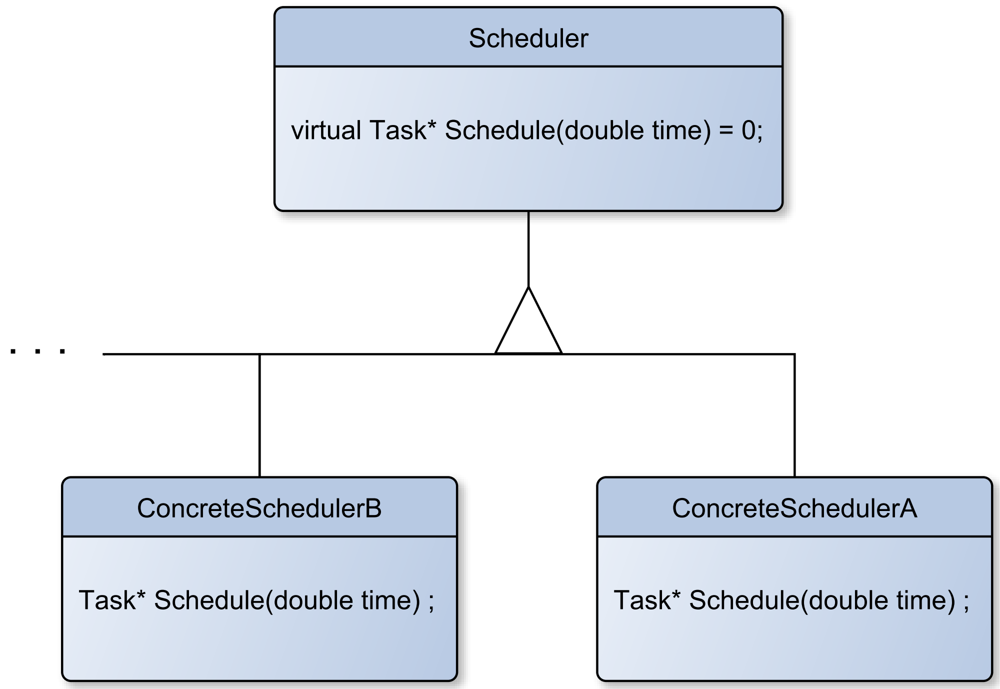

Scheduler
virtual Task* Schedule(double time) = 0;
. . .
ConcreteSchedulerB
Task* Schedule(double time) ;
ConcreteSchedulerA
Task* Schedule(double time) ;

Figure 4.3: Structure of the abstract Scheduler class with sub-classes

Based on the system created in the second iteration shown in figure 4.2, changes were made to accommodate the new `Scheduler` class. A Simulator class was added to contain the behavior of the simulated CPU. The resulting system is shown in figure 4.4

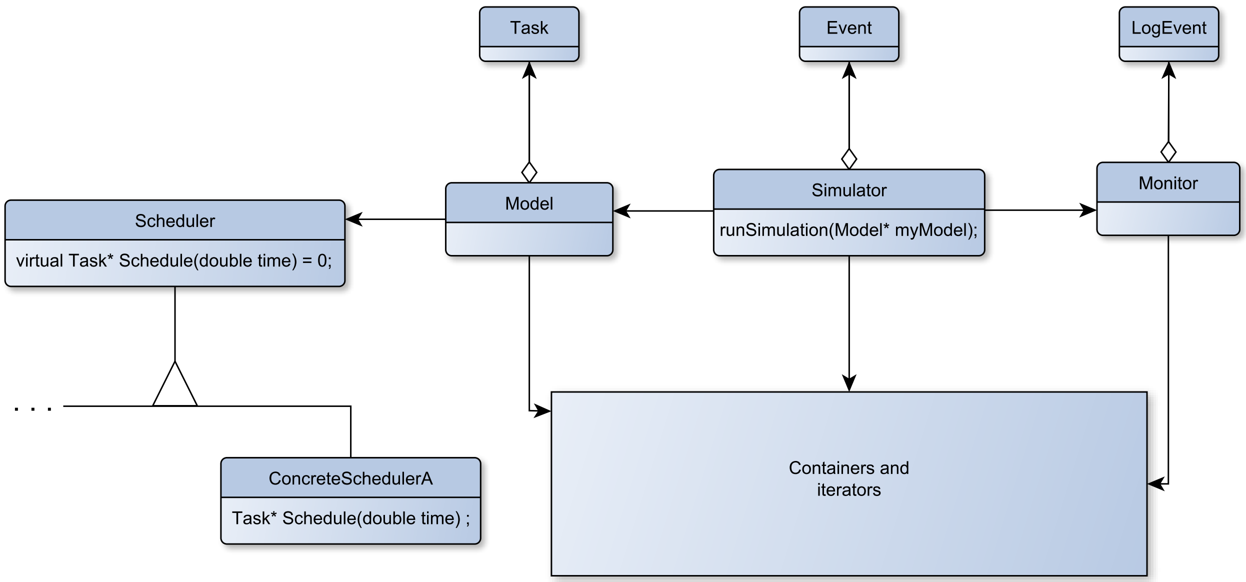

Task
Event
LogEvent
Scheduler
virtual Task* Schedule(double time) = 0;
Model
Simulator
runSimulation(Model* myModel);
Monitor
. . .
ConcreteSchedulerA
Task* Schedule(double time) ;
Containers and iterators

Figure 4.4: A simplified Class Diagram of the Scheduler Simulator after the third iteration

`The Model class` contains a Scheduler, a Set of Tasks and all additional information needed to create a simulation, such as specified simulation run time and clock interval time.

`The Simulator class` executes the simulation of a `Model` object. The simulation is event-based, i.e. the current event decides the action taken by the simulator.

`Events` are created and stored in a Queue by the `Simulator`.

`The Monitor class` is responsible for logging the behavior of the simulator and generating relevant reports.

`LogEvents` are stored in a Queue by the `Monitor`.

`Task` sets, `Event` queues and `LogEvent` queues are all object containers. Instead of creating three designated classes to accommodate their almost equivalent needs, a design pattern was used. The Iterator pattern, see section 2.3, gives the opportunity of creating reusable containers with corresponding iterators, providing a protective layer for the content of the containers. Inspired by the Iterator pattern with structure shown in figure 2.12, the container-iterator system shown in figure 4.5, was created.

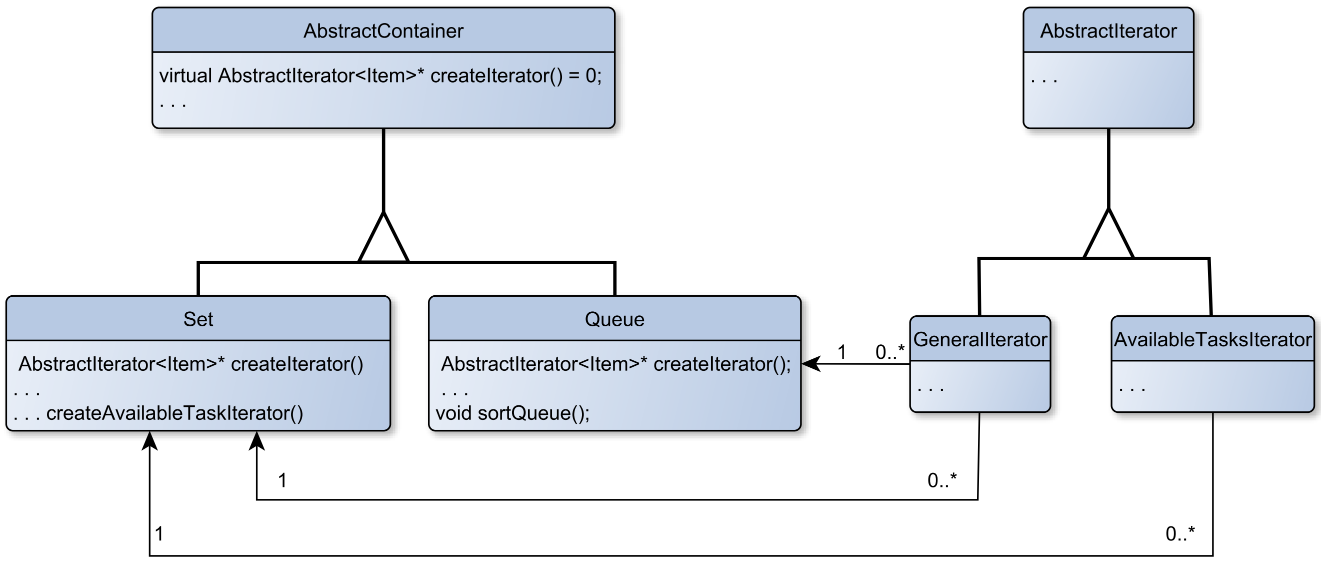

AbstractContainer
virtual AbstractIterator<Item>* createIterator() = 0;
. . .
AbstractIterator
. . .
Set
AbstractIterator<Item>* createIterator()
. . .
. . . createAvailableTaskIterator()
Queue
AbstractIterator<Item>* createIterator();
. . .
void sortQueue();
GeneralIterator
. . .
AvailableTasksIterator
. . .
1
0..*
1
0..*
1
0..*

Figure 4.5: Simplified class diagram of the container and iterator system, based on the "Iterator design pattern"

`The AbstractContainer class` is an abstract class, providing the main interface for interaction with the sub-container classes.
`The Set class` is a container for storing objects when the order is insignificant.
`The Queue class` is a container for storing objects. The `Queue` can be sorted.

`The AbstractIterator class` is an abstract class, providing the interface for interaction with the sub-iterator classes.
`The GeneralIterator class` represents an iterator that can traverse both a `Set`-object and a `Queue`- object.
`The AvailableTaskIterator class` represents an iterator traversing a `Set`- object storing Task pointers. The `AvailableTaskIterator` returns only available tasks, i.e. `Task`-objects ready for execution.

What was learned from this iteration: The design patterns provide reusable and low maintainable code. Separating responsibility between designated classes ensures and helps achieving correct behavior.

Fourth Iteration: Simulator for all Schedulers with Generated Task Sets

The fourth iteration incremented the amount of requirements to be met, thus including all main requirements relevant for architectural aspects.

Requirements:

- The software simulates a scheduler and a task set.
- The output of the system consists of relevant plots and graphic representations of the results.
- The simulator is openly created to simulate new scheduling types.
- The simulator is able to generate task sets.
- The user is able to chose the distribution of the generated task sets.

To be able to generate task sets, a `TaskHandler` class was added. The `TaskHandler` contains task-relevant functionality needed by the `Model` and `Simulator` classes. The `TaskHandler` is used to generate task sets inside a `Model` object. This relationship is illustrated in figure 4.6.

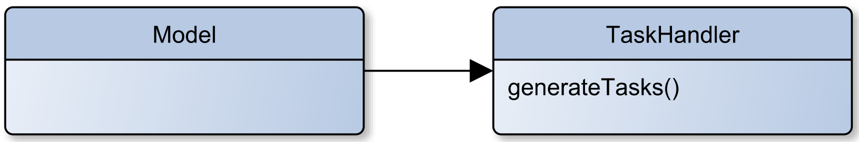


Figure 4.6: Simplified Class Diagram. The TaskHandler class is an object within the Model class

The final class diagram for the Scheduling Simulator is shown in figure 6.4.

4.2.4 Class Descriptions and Design Choices

The classes and their functionality were created and evolved during the iterations described in section 4.2.3. This section describes and explains the class design choices made by the developer. The final design and architectural documentation can be found in section 6.2.

The `Task` Class

A Task object represents a processes or an executable thread. Based on the simplified task life span state diagram in figure 3.1 and section 3.1.2, a set of states were specified for the task class, listed in table 6.2.
The `Task` data members were created based on their relevance for simulating a scheduler. The existing scheduling policies listed in section 3.1.3, uses *arrival time*(FIFO), *deadline*(EDF), *period*(RM) and *estimated execution time*(SJF) to make their decisions. Additional Task information may be relevant for other schedulers, such as the Tasks overall *progression* of the execution. Based on these factors, the `Task` data members listed in table 6.3 were created.
To protect some of these variables from modification by other classes, all data members, except for `priority`, were set as `private` members. Simple `get-` functions were implemented for these data members, giving others, such as the Scheduler class, a read-only access. The `priority` member is created for the sake of the Scheduler, and is thus both read- and writable for the `Scheduler`.
Periodic `Task` objects contain a non-zero `period` value, giving the Tasks a cyclic execution behavior, whereas aperiodic `Task` objects' `periods` are equal to zero, and they are only executed one time. The complete `Task` class can be seen in figure 6.5.

The `Model` Class

The `Model` class was created to contain all the parameters needed to conduct a simulation. This way, several `Model` objects can be simulated by the same `Simulator` object. The main data members of the `Model` are a pointer to a task set to be executed, and a pointer to a `Scheduler` object. Other parameters address the simulation run time, which is default set to 150 ms, and the clock cycle, which is default set to be 1 ms. The `Model` data members are listed in table 6.4.
The `Model` is implemented with multiple constructors, giving the user several possibilities to create a `Model` object and set its variables. The multiple constructors make the user choose to either create a task set manually and give it to the `Model`, or have the `Model` generate a task set itself. The different `Model`-constructors are listed in table 6.5. The complete `Model` class can be seen in figure 6.6. A string with a model name must be given to the `Model` object, to later be used as a name for the resulting simulation report generated by the `Monitor` object.

The `TaskHandler` Class

The `TaskHandler` class is used by the `Model` class. The `TaskHandler` class is, among other things, responsible for generating `Task` objects. The generation of `Task` objects are based on a normal distribution. The TaskHandler can generate both periodic an aperiodic `Task` objects, and distributes them accordingly.

The TaskHandler handles the Model object's Task set, and executes Task-related functions during simulation, such as updating Task states corresponding to the simulation progress and discovering missed deadlines. The complete TaskHandler class can be seen in figure 6.7.

The Event Class

The Event class is utilized by the Simulator class. Event-objects consists of an EventTime and an EventType. The EventType represents events happening during the simulation, and the EventTime is the time of the event. The different types of events are based on whenever a scheduler might have to make a decision. The EventTypes are listed in table 6.6. The complete Event class can be seen in 6.9.

The Simulator Class

The Simulator class represents a simulated processor, executing Task objects based on a scheduling decision. The Simulator class simulates Model objects, thus one Simulator object can run simulations of several Model objects.

SIMULATION BEHAVIOR:
The simulation is event-driven. Events are queued, and the event closest in time is read from the queue. The Simulator responds to the current event with a corresponding action. Events are created and added to the queue at different occasions:

- **At time zero:** A SimulationFinished event is created.
- **At TimeInterrupt:** A new TimeInterrupt event is created for *preemptive* Scheduler- objects.
- **When no Task is available:** a TaskReady event is created for the future when a task becomes available for execution.
- **When a Task is being executed:** a TaskFinished event is created for the future, for when the task ideally finishes its execution.

If a Task currently being executed is interrupted, its corresponding TaskFinished event will be deleted from the Event queue. When a SimulationFinished event is reached, relevant reports are generated, and the Simulator object is free to simulate other Model-objects. This form of event-driven behavior allows the Simulator to only react and respond when needed. The complete Simulation class can be seen in 6.8. A flow-chart of the Simulators behavior is shown i figure 6.16.

The `Monitor` Class

The `Monitor` class logs relevant events and generates a report at the end of each simulation. During a simulation, a `Monitor`- objects keeps track of a queue of `LogEvent`-objects, adding `LogEvents` when needed. When the simulation is finished, the queue is traversed and a report is generated based on the chain of events. The report is created using LaTeXand the rtsched LaTeXstyle to generate a pdf format file, 3.3.3. To mark deadlines specifically, a new function was added to the rtsched LaTeXstyle, called `DeadlineMissed`. The name of the resulting pdf file is set to be the string name of the corresponding simulated `Model`-object. Relevant output information from the simulation were considered to be: information about the simulated task set, resulting CPU behavior and deadline misses. Visual presentation of the `Task` executions is provided using Gantt charts, as they are easy to understand and they include a time domain representation. The average turnaround time (see section 3.1.3) is included in the resulting reports to measure the efficiency of the simulated scheduler. Information to directly compare the performance of schedulers is useful, such as the CPU utilization, see section 3.1.1. The complete list of elements presented in the output report can be seen in table 6.1.
The complete `Monitor` class is shown in figure 6.10.

The `LogEvent` Class

The `LogEvent` class is used by the `Monitor` class to store relevant events during the simulation. The `LogEvent` class has three attributes, which are the *time* of the event, the *EventType* and the event related *Task*. The different `EventTypes`, listed in table 6.7 were created to cover all the information needed to generate the wanted report content post-simulation. The complete `LogEvent` class can be seen in figure 6.11.

The `Scheduler` Class

The `Scheduler` class is a base class for scheduling policies. New scheduling classes can inherit this base class to implement their scheduling behavior. The `Scheduler` base class shall provide an open environment to permit implementation of existing as well as future scheduling policies. Considering the existing scheduling policies, the matter of *preemption* is essential, see section 3.1.3. Letting the scheduler itself state whether or not it has preemptive behavior enables the `Simulator` class to only ask for scheduling decisions correspondingly. This can simplify the implementation process for a new scheduler, as the scheduler logic does not need to consider the circumstances at the time it is asked to make a decision. A preemptive scheduler is called upon periodically, regardless of a task is currently being executed or not, while a non-preemptive scheduler is only called upon when no task is being executed. Future schedulers may not have this strict distinction, but using the preemptive version enables great freedom for the new scheduler to implement its own logic by filtering out unwanted scheduling calls.

The scheduler logic is decided by the implementation of the virtual function `"Task* schedule(double time)"`. This function must be implemented in the new scheduling sub-classes. This function takes the simulation progression time as an argument, and returns a pointer to the optimal `Task` object to execute, according to the scheduler policy. A vast variations of schedulers can implement this function to contain their own logic. Additional internal functions and variables can be implemented in the new scheduling sub-class to assist the logic in their `scheduling` function.

The `Scheduler` object should be able to access relevant `Task` data members, but should not be able to change them, with one exception; the `Task`'s priority. This problem is solved in the implementation of the `Task` class, using get-functions, see section 6.2.2.
The `Scheduler` is not allowed to add new `Task` objects to the system. Using `iterators` to traverse the task set provides a protective layer, giving the `Scheduler` class a "read-only" access to the task set. As opposed to a queue of `Task` objects in the state `READY`, an iterator is provided to only return available `Task` and filter out `Task` objects in states indicating that they are not ready for execution. Future scheduling strategies may need to consider all `Task` objects in the system, and for this purpose another iterator is added, traversing the entire task set, regardless of the `Task` objects state. So, The `Scheduler` class can see all tasks in the system, but can only let the `"Task* schedule(double time)"` return a task from the filtered iterator. The complete `Scheduler` class can be seen in figure 6.12. Currently, four sub-classes of the Scheduler class are implemented:

- **FIFO**
- **RR**
- **RM**
- **EDF**

The Container and Iterator Classes

Based on the *Iterator* design pattern, 2.3, a reusable system of containers and iterators were created. An iterator provides a protective layer between the user and the contained objects. The two base classes `AbstractContainer` and `AbstractIterator` provide the interface for interactions with the containers and the iterators. Subclasses of the AbstractContainer class can add additional functionality to meet specific needs. Two different containers were created:

`The Set class` is primarily created for sets of unordered objecs, where accessing an object at a specific index is important. The `Set` class uses a `std::vector` to store objects. `Task` objects are stored in a `Set` container, creating a task set.

`The Queue class` is created to accommodate sorted queues of objects, only needing the first, and optimal, element in the queue. `Event` objects and `LogEvents` are stored in `Queue` objects.

A general iterator class, called `GeneralIterator`, was created to traverse the objects stored in both `Set` objects and `Queue` objects.
A filtered iterator class, called `AvailableTaskIterator`, was created specific for `Set` objects containing `Task` objects. The `AvailableTaskIterator` points only to `Task` objects available for execution, i.e. tasks in the states READY, RUNNING or WAITING.

The complete system of containers and iterators can be seen in section 6.2.10.

4.2.5 Creating Documentation

The documentation was created during the development of the software, and was updated when changes were made. In section 2.2, key aspects involving documentation are pointed out. Emphasizing Agile development methodologies, slim, sufficient and readable documentation was desirable to produce for this software project.

Requirement Documentation

Requirement documentation was created using the theory described in section 2.2.1. Written, precise requirements were created, describing the main goals of the software. To supplement, User Stories were created based on the written requirements. Use Cases were created to portrait the interactive behavior of the software. Two Use Cases were created to illustrate the user-software interaction, and one Use Case was created to illustrate the internal behavior of the software. The requirement documentation can be seen in section 6.1.

Architecture and design

The architecture and design of the software was documented, and continuously updated during the development, evolving into an up-to-date description of the structure and relations of the classes in the `Scheduler Simulator`. Using written words, tables and diagrams, the architecture was documented to enable another person to read and understand the bits and pieces of the created software without too much unnecessary descriptions. The goal of this documentation was to present the software design without too much explanations of why these design choices were made. As stated in section 2.2.2, using diagrams may provide a good overview and help conveying the information. Based on this information, class diagrams were used to illustrate the different classes and the design of the software. A simplified class diagram presenting the entire system was deliberately placed in the beginning of the documentation to provide the reader with an overview before proceeding to read detailed descriptions of each class. The architectural documentation can be seen in section 6.2

User Documentation

The user documentation consists of two main parts: the final-documentation and the end-user documentation (section 2.2.3).

The final-documentation is meant to provide enough information for a maintenance personnel to operate the system when launched. Providing explanatory comments, referencing to the architectural documentation, referencing to code documentation, explaining shortcomings and external dependencies seems to fulfill this demand. Good structure and readability were emphasized when creating the final-documentation, using a natural and unambiguous language. The final-documentation can be seen in section 6.3.

The end-user documentation must provide enough information for a user to utilize the software as intended by the developing team. Too much information may confuse and unnecessarily force the user to learn about functionality he or she indirectly interact with. For this software, the user needs to be able to do two operations:

- Run a simulation.
- Add a scheduling policy.

An appealing way to do this is descriptive tutorials, walking the user step by step through the needed actions to make these two exact things happen. The tutorials will enable the user to expand and utilize the software. Using tutorials, only the necessary amount of information is shown to the end-user. The end-user documentation, containing tutorials for using the Scheduler Simulator, can be found in section 6.4.

4.3 Test Runs of the Scheduling Simulator

To test the limits and the functionality of the finished developed Scheduler Simulator software, a series of tests were conducted. All the tests were executed by a single run of the Scheduler Simulator program, using one `Simulator` object to simulate six `Model` objects.

4.3.1 Simulate Specified Task Set

Using a specified Task set, created manually, all the four implemented scheduling policies were simulated. The simulation run time was 200(ms), clock cycle and context switch penalty were default set to 1 ms and 0 ms, respectively. The simulated Task set can be seen in table 4.2.

Table 4.2: Manually created Task Set

Task number	Arrival Time	Execution Time	Deadline	Period
1	0	24	100	100
2	0	20	80	80
3	0	20	60	60

The simulated schedulers were:

- RM scheduler.
- FIFO scheduler.
- RR scheduler.
- EDF scheduler.

The results can be seen in section 5.1.

4.3.2 Simulate Generated Task Set

Using the EDF scheduler, two test runs were conducted with generated task sets. The simulation test runs are presented in table 4.3. The amount of tasks was varied to discover limitations in the capacity of the simulator.

Table 4.3: Simulation runs with Generated Task Sets

Simulation number	# Periodic Tasks	# Aperiodic Tasks	Run-Time
1	5	5	100
2	10	10	100

The results from these test can be seen in section 5.2.

Chapter 5

Simulation Test Results

5.1 Results from Test Runs with Specified Task Set

The test runs described in section 4.3.1 generated the following reports.

5.1.1 RM Simulation Report

Simulation of Scheduling in Real-Time Systems Model:
RMManuallyTaskSet

Table 1: Tasks

Task number	Arrival Time	Execution Time	Deadline	Period
1	0	24	100	100
2	0	20	80	80
3	0	20	60	60

Table 2: Average Turnaround Times

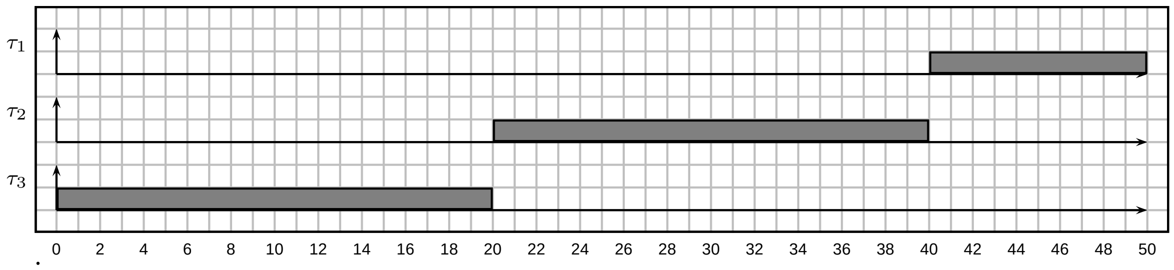
Task	Average Turnaround Time
1	74
2	26.6667
3	20
Total Average Turnaround Time	40.2222

Table 3: Simulation Results in Numbers

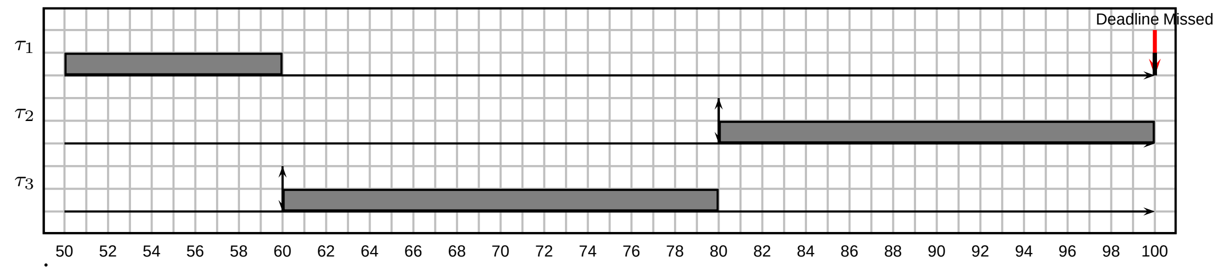
CPU Idle Time	12
Number of Deadlines reached	8
Number of Deadlines missed	1
Number of Completed tasks	3
Number of Tasks	3

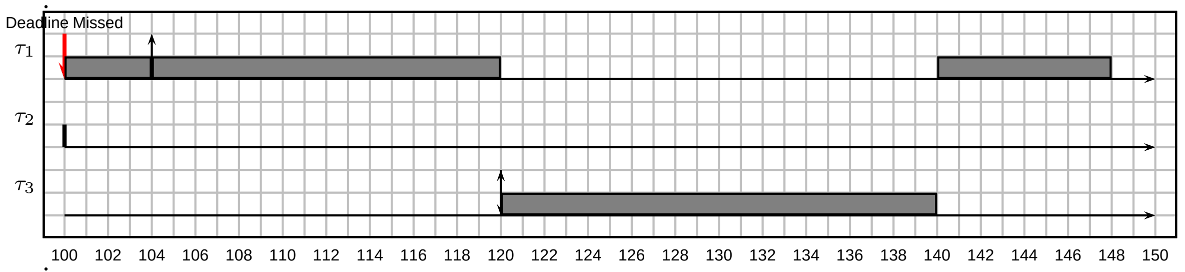
Table 4: Simulation Results in Percentage

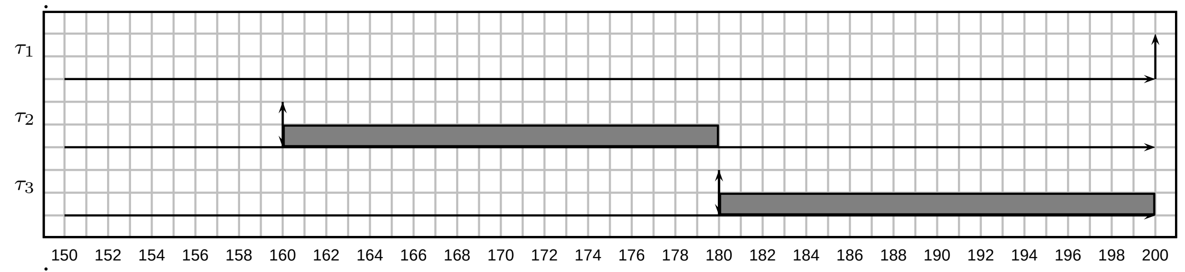
CPU Idle Time	6%
Deadlines missed	11.1111%
Completed tasks	100%



.
.


Deadline Missed
τ_1
τ_2
τ_3
50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100


Deadline Missed
τ_1
τ_2
τ_3
100 102 104 106 108 110 112 114 116 118 120 122 124 126 128 130 132 134 136 138 140 142 144 146 148 150


τ_1
τ_2
τ_3
150 152 154 156 158 160 162 164 166 168 170 172 174 176 178 180 182 184 186 188 190 192 194 196 198 200

5.1.2 FIFO Simulation Report

Simulation of Scheduling in Real-Time Systems Model:
FIFOManuallyTaskSet

Table 1: Tasks

Task number	Arrival Time	Execution Time	Deadline	Period
1	0	24	100	100
2	0	20	80	80
3	0	20	60	60

Table 2: Average Turnaround Times

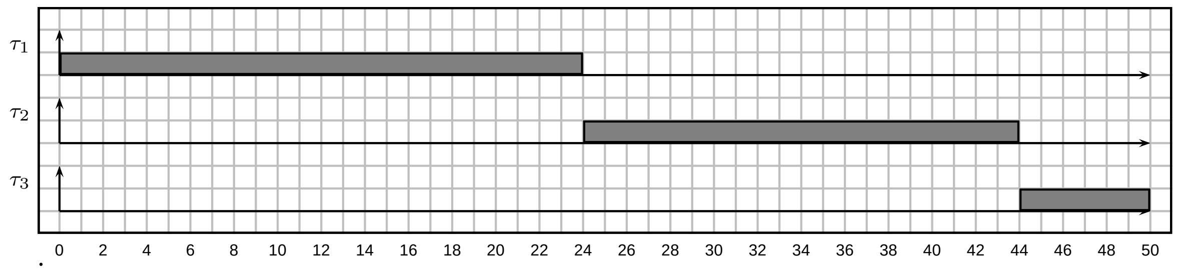
Task	Average Turnaround Time
1	26
2	29.3333
3	33
Total Average Turnaround Time	29.4444

Table 3: Simulation Results in Numbers

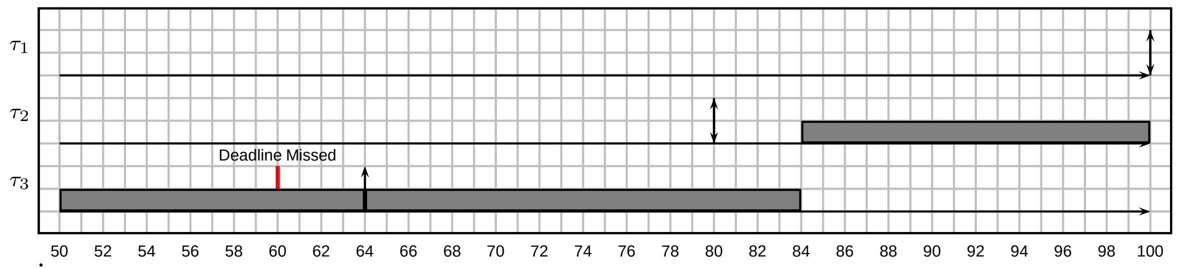
CPU Idle Time	12
Number of Deadlines reached	8
Number of Deadlines missed	1
Number of Completed tasks	3
Number of Tasks	3

Table 4: Simulation Results in Percentage

CPU Idle Time	6%
Deadlines missed	11.1111%
Completed tasks	100%



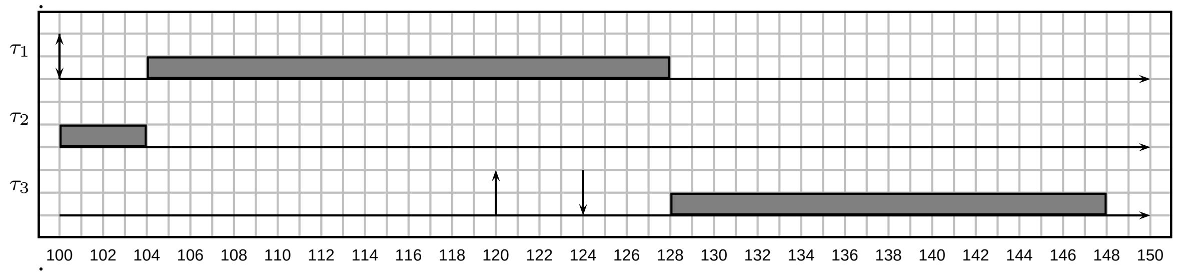
.
.



τ_1
τ_2
τ_3
Deadline Missed
50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100

.

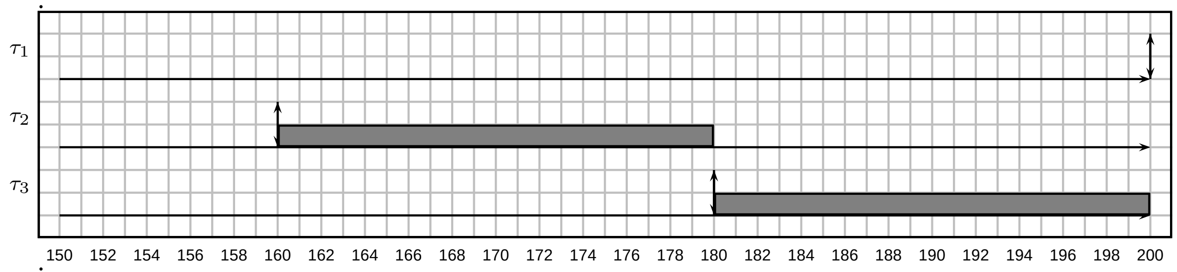
.



τ_1
τ_2
τ_3
100 102 104 106 108 110 112 114 116 118 120 122 124 126 128 130 132 134 136 138 140 142 144 146 148 150

.

.



τ_1
τ_2
τ_3
150 152 154 156 158 160 162 164 166 168 170 172 174 176 178 180 182 184 186 188 190 192 194 196 198 200

.

.

5.1.3 RR Simulation Report

Simulation of Scheduling in Real-Time Systems Model:
RRManuallyTaskSet

Table 1: Tasks

Task number	Arrival Time	Execution Time	Deadline	Period
1	0	24	100	100
2	0	20	80	80
3	0	20	60	60

Table 2: Average Turnaround Times

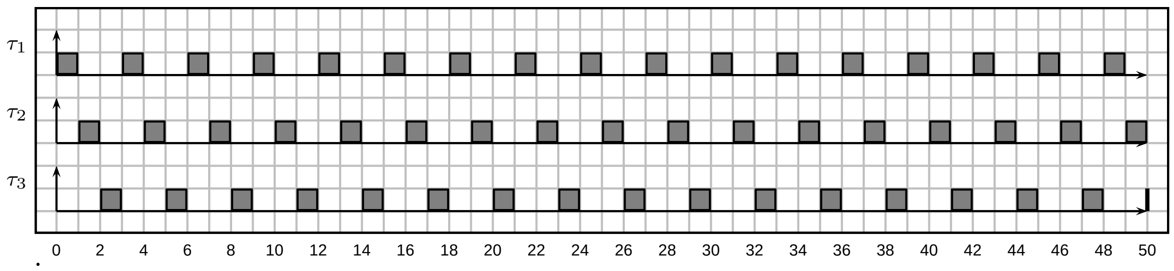
Task	Average Turnaround Time
1	51.5
2	36.6667
3	34.25
Total Average Turnaround Time	40.8056

Table 3: Simulation Results in Numbers

CPU Idle Time	12
Number of Deadlines reached	8
Number of Deadlines missed	1
Number of Completed tasks	3
Number of Tasks	3

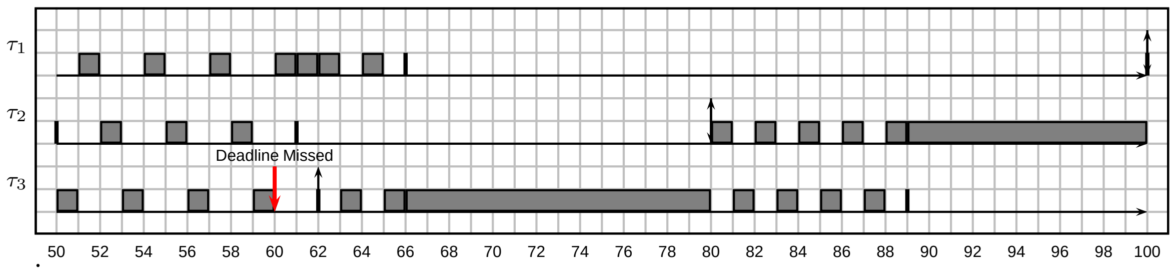
Table 4: Simulation Results in Percentage

CPU Idle Time	6%
Deadlines missed	11.1111%
Completed tasks	100%



.
.

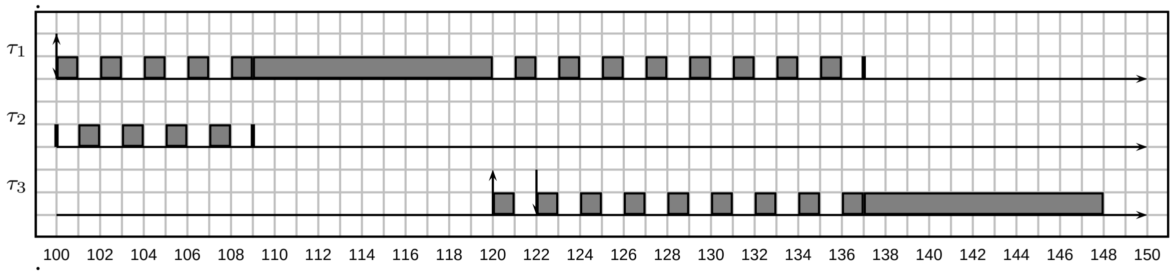
76



τ_1
τ_2
Deadline Missed
τ_3
50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100

.

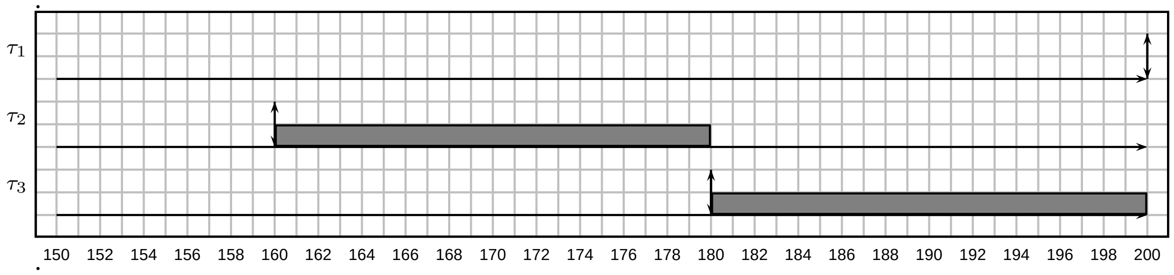
.



τ_1
τ_2
τ_3
100 102 104 106 108 110 112 114 116 118 120 122 124 126 128 130 132 134 136 138 140 142 144 146 148 150

.

.



τ_1
τ_2
τ_3
150 152 154 156 158 160 162 164 166 168 170 172 174 176 178 180 182 184 186 188 190 192 194 196 198 200

.

.

5.1.4 EDF Simulation Report

Simulation of Scheduling in Real-Time Systems Model:
EDFManuallyTaskSet

Table 1: Tasks

Task number	Arrival Time	Execution Time	Deadline	Period
1	0	24	100	100
2	0	20	80	80
3	0	20	60	60

Table 2: Average Turnaround Times

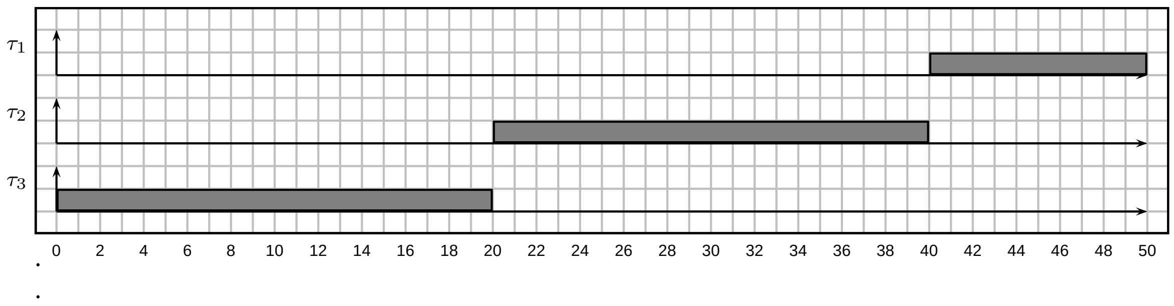
Task	Average Turnaround Time
1	56
2	28
3	21
Total Average Turnaround Time	35

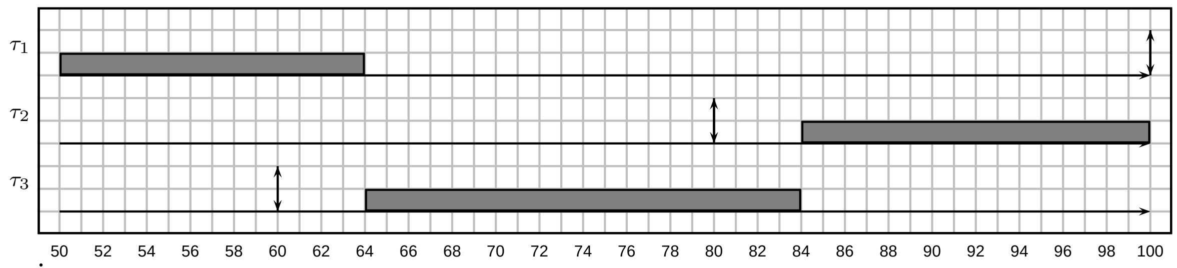
Table 3: Simulation Results in Numbers

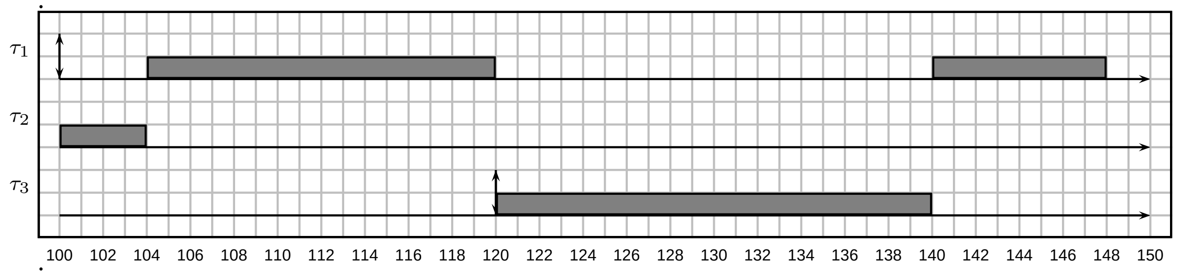
CPU Idle Time	12
Number of Deadlines reached	9
Number of Deadlines missed	0
Number of Completed tasks	3
Number of Tasks	3

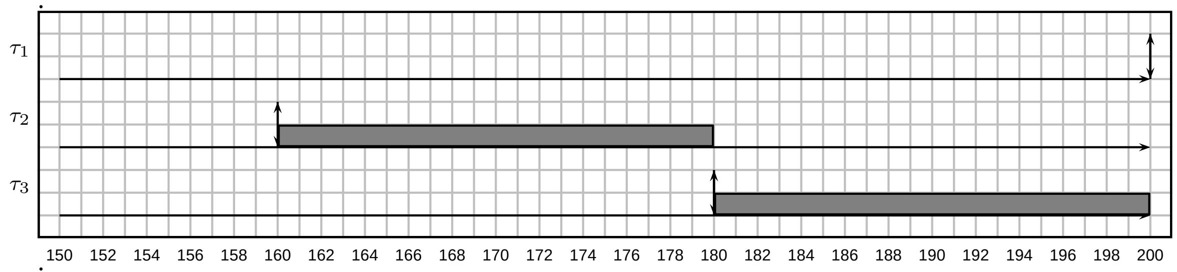
Table 4: Simulation Results in Percentage

CPU Idle Time	6%
Deadlines missed	0 %
Completed tasks	100%




τ_1
τ_2
τ_3
50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100


τ_1
τ_2
τ_3
100 102 104 106 108 110 112 114 116 118 120 122 124 126 128 130 132 134 136 138 140 142 144 146 148 150


τ_1
τ_2
τ_3
150 152 154 156 158 160 162 164 166 168 170 172 174 176 178 180 182 184 186 188 190 192 194 196 198 200

5.2 Results from Test Run with Generated Task Set

The test runs described in section 4.3.2 generated the following reports.

5.2.1 EDF Simulation Report with Ten Generated Tasks

Simulation of Scheduling in Real-Time Systems Model:
EDF10GeneratedTasks100

Table 1: Tasks

Task number	Arrival Time	Execution Time	Deadline	Period
1	2	69	207	207
2	3	137	411	411
3	0	82	246	246
4	0	18	54	54
5	6	2	6	6
6	0	69	207	Aperiodic
7	0	137	411	Aperiodic
8	0	82	246	Aperiodic
9	0	18	54	Aperiodic
10	0	2	6	Aperiodic

Table 2: Average Turnaround Times

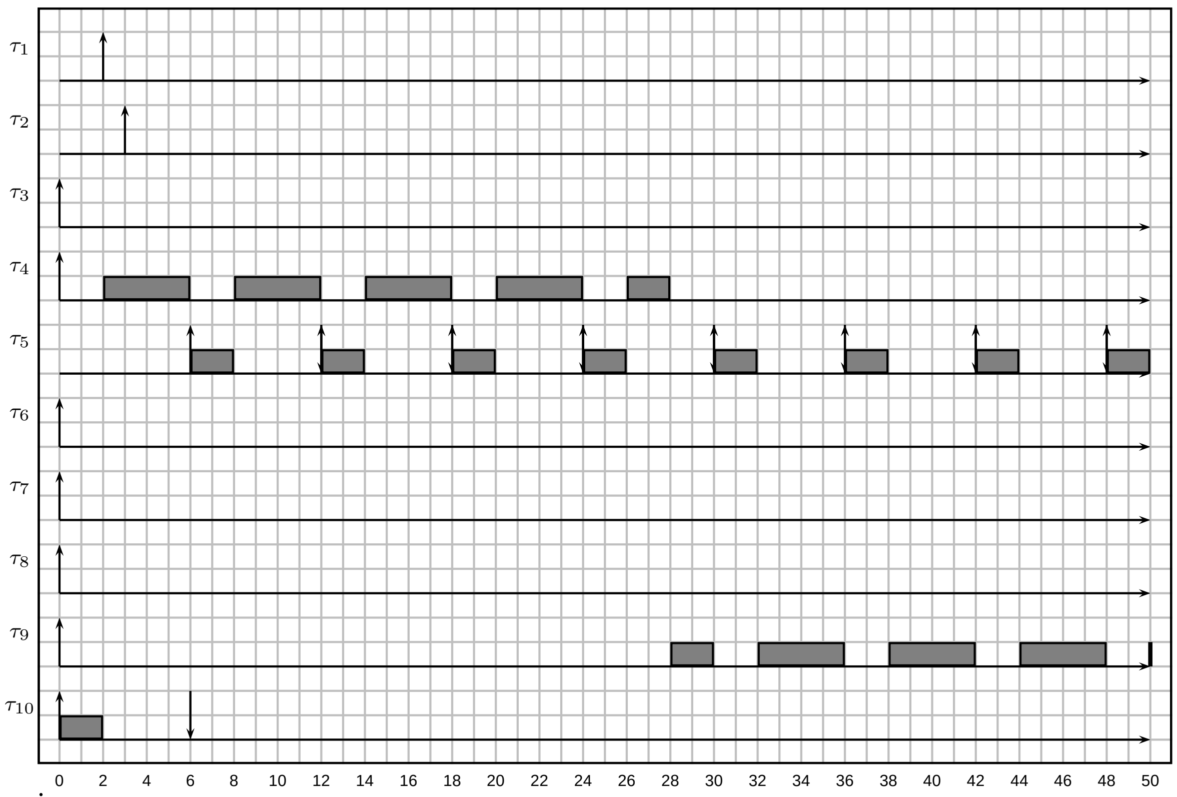
Task	Average Turnaround Time
1	No Value
2	No Value
3	No Value
4	28
5	2
6	No Value
7	No Value
8	No Value
9	54
10	2
Total Average Turnaround Time	21.5

Table 3: Simulation Results in Numbers

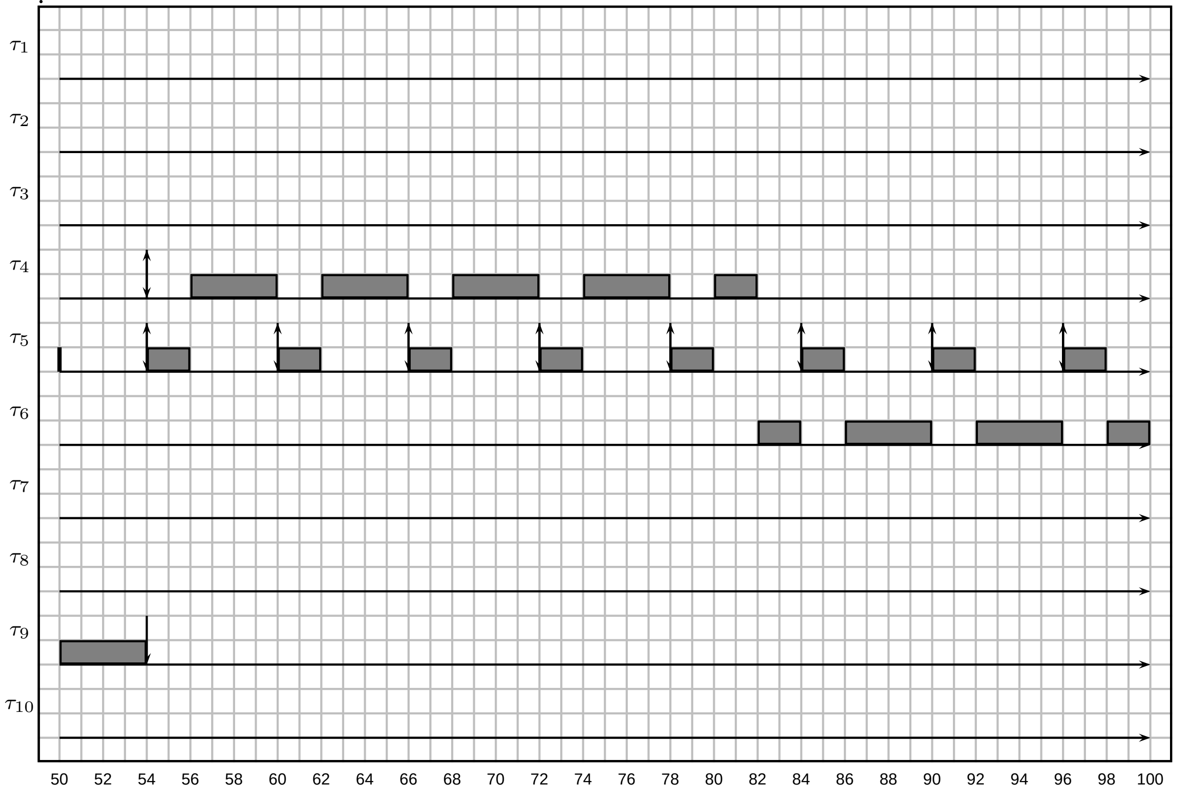
CPU Idle Time	0
Number of Deadlines reached	20
Number of Deadlines missed	0
Number of Completed tasks	4
Number of Tasks	10

Table 4: Simulation Results in Percentage

CPU Idle Time	0%
Deadlines missed	0 %
Completed tasks	40%



τ_1
τ_2
τ_3
τ_4
τ_5
τ_6
τ_7
τ_8
τ_9
τ_{10}
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50



τ_1
τ_2
τ_3
τ_4
τ_5
τ_6
τ_7
τ_8
τ_9
τ_{10}
50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100

5.2.2 EDF Simulation Report with Twenty Generated Tasks

Simulation of Scheduling in Real-Time Systems Model:
EDF20GeneratedTasks100

Table 1: Tasks

Task number	Arrival Time	Execution Time	Deadline	Period
1	2	9	27	27
2	0	37	111	111
3	0	90	270	270
4	5	153	459	459
5	0	192	576	576
6	0	180	540	540
7	8	129	387	387
8	0	71	213	213
9	0	30	90	90
10	11	10	30	30
11	12	9	27	Aperiodic
12	0	37	111	Aperiodic
13	14	90	270	Aperiodic
14	0	153	459	Aperiodic
15	16	192	576	Aperiodic
16	0	180	540	Aperiodic
17	18	129	387	Aperiodic
18	19	71	213	Aperiodic
19	20	30	90	Aperiodic
20	0	10	30	Aperiodic

Table 2: Average Turnaround Times

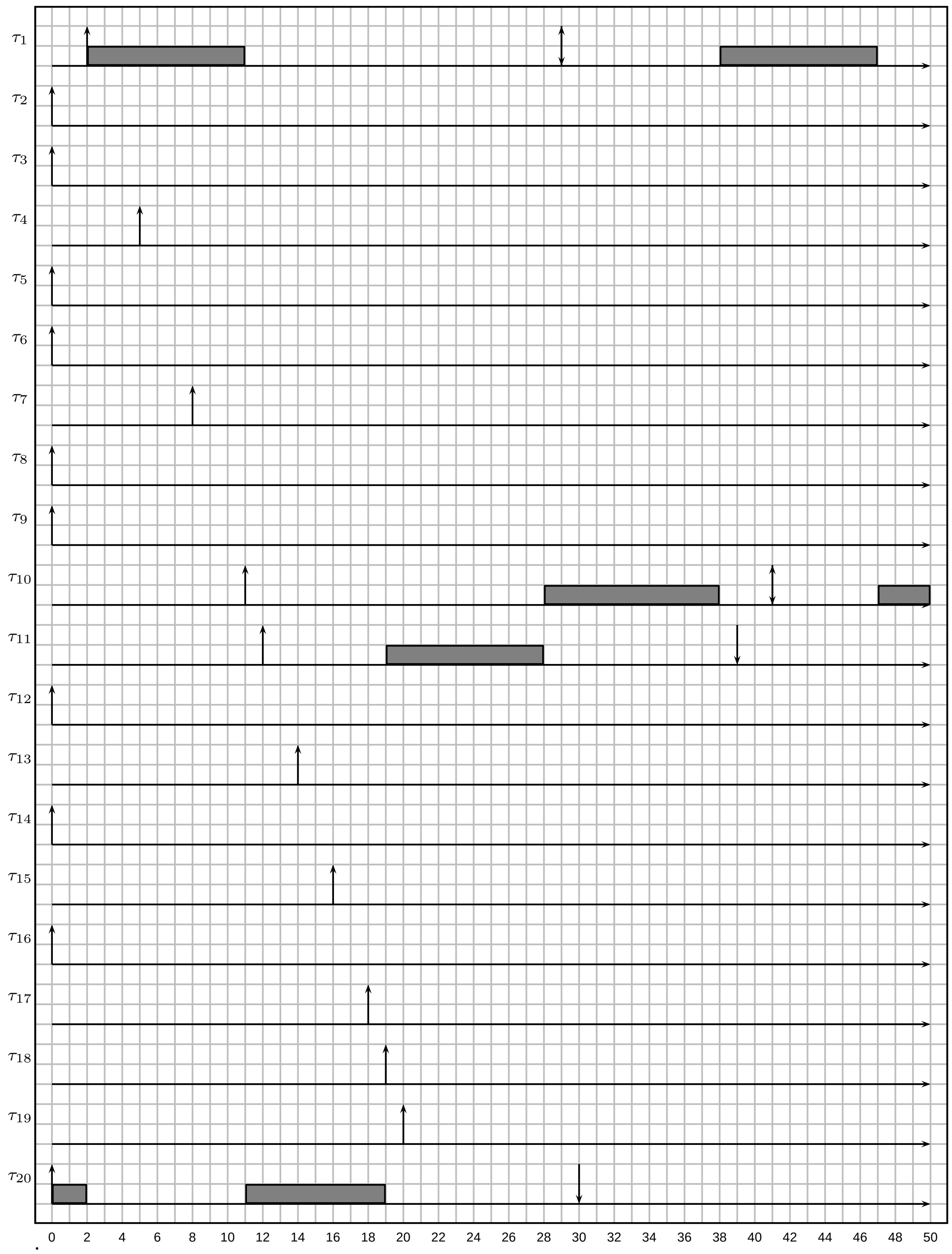
Task	Average Turnaround Time
1	12.3333
2	No Value
3	No Value
4	No Value
5	No Value
6	No Value
7	No Value
8	No Value
9	96
10	21.5
11	16
12	No Value
13	No Value
14	No Value
15	No Value
16	No Value
17	No Value
18	No Value
19	No Value
20	19
Total Average Turnaround Time	32.9667

Table 3: Simulation Results in Numbers

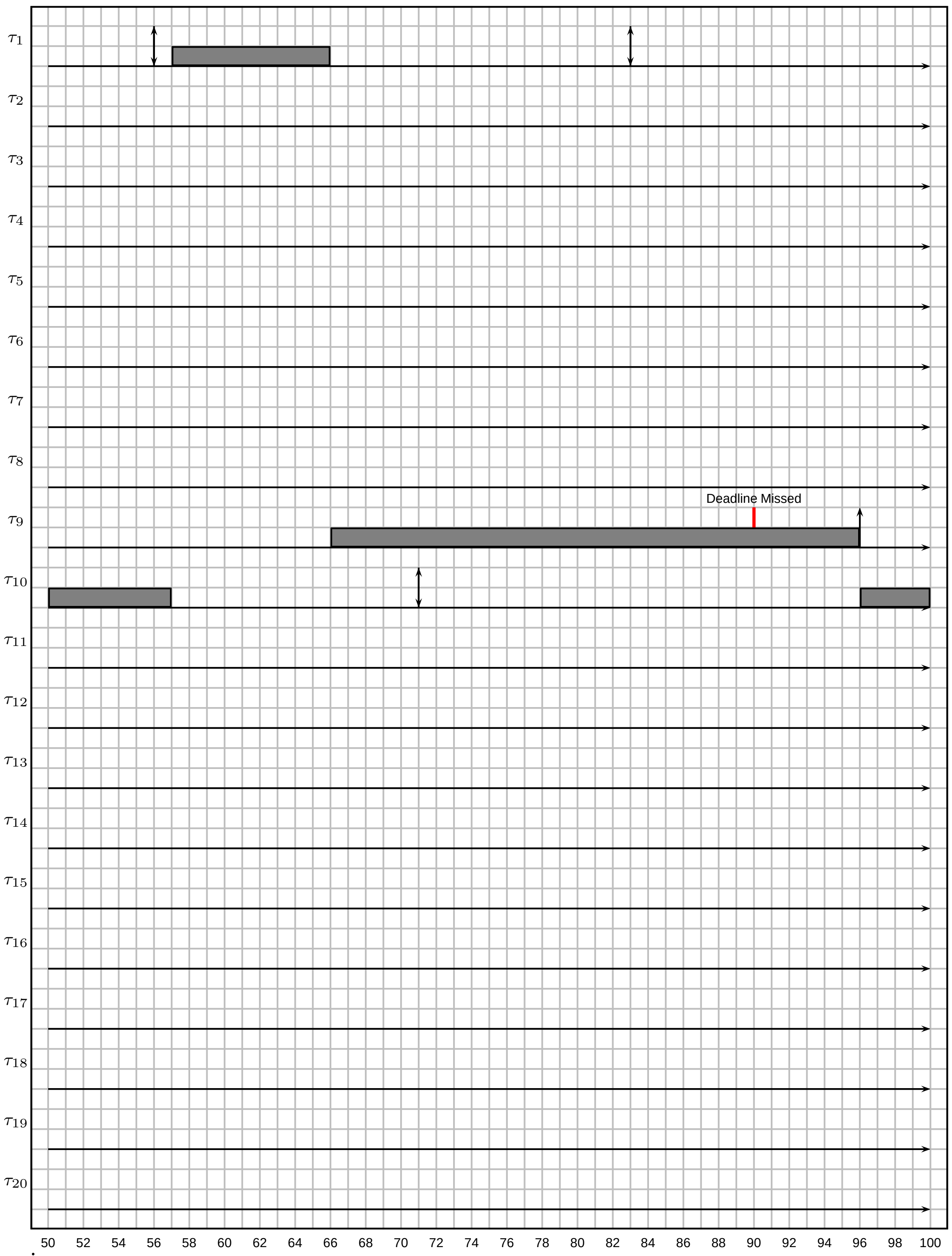
CPU Idle Time	0
Number of Deadlines reached	7
Number of Deadlines missed	1
Number of Completed tasks	5
Number of Tasks	20

Table 4: Simulation Results in Percentage

CPU Idle Time	0%
Deadlines missed	12.5%
Completed tasks	25%


τ_1
τ_2
τ_3
τ_4
τ_5
τ_6
τ_7
τ_8
τ_9
τ_{10}
τ_{11}
τ_{12}
τ_{13}
τ_{14}
τ_{15}
τ_{16}
τ_{17}
τ_{18}
τ_{19}
τ_{20}
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50

.
.



τ_1
τ_2
τ_3
τ_4
τ_5
τ_6
τ_7
τ_8
τ_9
τ_{10}
τ_{11}
τ_{12}
τ_{13}
τ_{14}
τ_{15}
τ_{16}
τ_{17}
τ_{18}
τ_{19}
τ_{20}
Deadline Missed
50 52 54 56 58 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98 100

Chapter 6

Documentation

This documentation is created to accompany the `Scheduling Simulator` software in C++. The documentation consists of the following parts:

- **6.1 Requirement Documentation:** Describes the specifications of the system, using text combined with Use Cases.
- **6.2 Architecture and Design Documentation:** Describes the design of the Scheduling Simulator using text and Class Diagrams.
- **User Documentation:** Consists of two main parts:
  - **6.3 Final Documentation:** Contains sufficient information and explanation enabling future maintenance personnel to operate the software.
  - **6.4 End-User Documentation:** Contains tutorials on how to use the different features of the Scheduling Simulator software.

6.1 Requirement Documentation

This section contains descriptions and illustrations of the requirements for a software to simulate the behavior of a processor given a scheduling policy.

Written Requirement Documentation for Scheduler Simulator

- The software can simulate different scheduling policies on different task sets.

- The user is a person, typically a student, familiar with the C++ language.
- The simulator is openly created to simulate new scheduling policies.
- New scheduling policies are easily added by the user.
- The input for the simulator consist of schedulers and task sets.
- The simulator shall generate distributed task sets.
- The output of the system shall consist of relevant plots and graphic representations of the results.
- A deadline miss must be clearly marked in the outputs.
- Statistical information to compare schedulers behavior shall be provided in the results.

6.1.1 User Stories

The following user stories describe specific scenarios that shall be possible to occur using the final product.

"The user shall simulate a FIFO scheduling policy and three tasks using the simulator."

"The Simulator provides a readable report after a simulation."

"The user implements a Round Robin scheduling algorithm and simulates it using the simulator."

6.1.2 Use Cases

Two use-cases were created to illustrate the users options to use the scheduling simulator: *"the user simulating a scheduler"* and *"the user creating a new scheduler"*. One use-case was created to illustrate the behavior of the simulator when running a simulation: *"The simulator simulating a scheduler"*.

User Simulating a Scheduler

A Use Case diagram for this use case can be seen in figure 6.1.

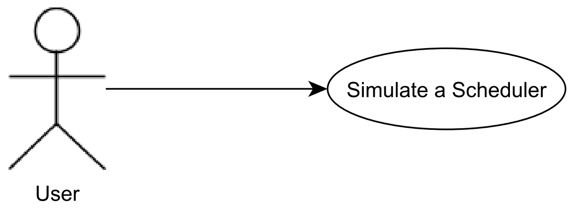


Figure 6.1: Use Case diagram for "user simulating a scheduler".

- **Use Case Name:** User simulates a scheduler using the simulator.
- **Iteration:** Focused.
- **Summary:** The user simulates a schedulers behavior to create relevant reports.
- **Basic Course of Events:**
  - 1. The user creates a model, consisting of a scheduler and a task set.
  - 2. The user simulates the model using the Simulator.
  - 3. Result report is accessible.
- **Alternative Paths:**
  - 1.a The user creates a task set.
  - 1.b The user generates a task set.
- **Exception Paths:** -
- **Extension Points:** State Machine "Simulator simulates schedulers behavior.", figure
- **Trigger:** A user wishes to view relevant reports of the behavior of a scheduler.
- **Assumptions:** The user knows how to create a model to simulate.

- **Preconditions:** The scheduler the user wishes to simulate must exist.
- **Postconditions:** The user must open the resulting simulation report.
- **Author:** Inger Johanne Rudolfsen
- **Date:**
  - Facade: 28. February 2015
  - Filled : 10.March 2015
  - Facade : 10.March 2015

User Implements Scheduling Policy

A Use Case diagram for this use case can be seen in figure 6.2.

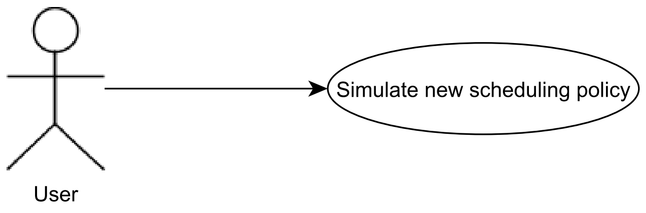


Figure 6.2: Use Case diagram for "user implementing a scheduling policy".

- **Use Case Name:** User Implements Scheduling Policy.
- **Iteration:** Focused.
- **Summary:** The user implements a new scheduler to be simulated by the scheduling simulator.
- **Basic Course of Events:**
  - 1. The user creates a scheduler.
  - 2. The user implements the scheduling policy.
  - 3. The user connects this scheduler to the rest of the software.
- **Alternative Paths:** -

- **Exception Paths:** -
- **Extension Points:** The use case: "User simulates a scheduler using the simulator"
- **Trigger:** A user wishes to simulate the behavior of a scheduler that is not yet implemented.
- **Assumptions:** The user knows how to implement a new scheduler.
- **Preconditions:** The user must know the logic behavior of the new scheduling policy.
- **Postconditions:** The new scheduler can be simulated.
- **Author:** Inger Johanne Rudolfsen
- **Date:**
  - Facade: 28. February 2015
  - Filled : 10.March 2015
  - Facade : 10.March 2015

Simulator running a simulation

A Use Case diagram for this use case can be seen in figure 6.3.

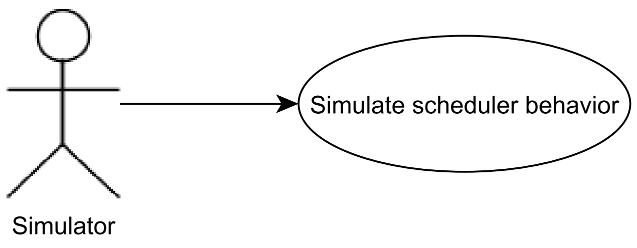


Figure 6.3: Use Case diagram for "Simulator simulates a Schedulers Behavior".

- **Use Case Name:** Simulator simulates a schedulers behavior.
- **Iteration:** Focused.

- **Summary:** Simulator simulates the behavior of a processors execution of a task set, using a specified scheduling policy.
- **Basic Course of Events:**
  - 1. Initialize.
  - 2. Create a Time Interrupt event.
  - 3. Get Event.
  - 4. Event is "Time Interrupt".
  - 5. create next "Time Interrupt" event.
  - 6. Ask Scheduler to Schedule.
  - 7. Scheduler returns a task.
  - 8. Task is set to RUNNING, and sets up a "Task Finished" event in the future.
  - 9. Return to 2.
- **Alternative Paths:** If event is not "Time Interrupt":
  - 4 b) If Event is "Task Finished"
  - 4 b) Set Task to "FINISHED".
  - 4 b) If Task is periodic, update Arrival Time.
  - 4 b) Go to point 6.

  - 4 c) If Event is "Simulation Finished"
  - 4 c) Generate Reports and Statistics.
  - 4 c) Finished
- **Exception Paths:** If the scheduler returns a task while a different task is being executed:
  - 7.b) Scheduler returns a task, while a different task is RUNNING.
  - 7.b) The RUNNING task must delete its "Task Finished" event.

  - 7.b) Go to point 8.

- **Extension Points:** The use case: "User simulates a scheduler using the simulator"

- **Trigger:** A simulator is asked by a user to simulate a model, consisting of a scheduler and a task set.

- **Assumptions:** The simulator receives a model, containing an existing scheduling policy.

- **Preconditions:** -

- **Postconditions:** Relevant reports are generated based on the event log.

- **Author:** Inger Johanne Rudolfsen

- **Date:**

  - Facade: 28. February 2015
  - Filled : 21.March 2015
  - Facade : 21.March

6.2 Architecture and Design Documentation

This section describes the classes in the Scheduling Simulator software, and their relationships, using various tables and class diagrams.

6.2.1 Scheduler Simulator System

The Scheduler Simulator software uses implemented scheduling policies to simulate task executions. A class diagram shown in figure 6.4 displays the entire system, class members excluded. More detailed descriptions of the classes exists in the next sections of this documentation.

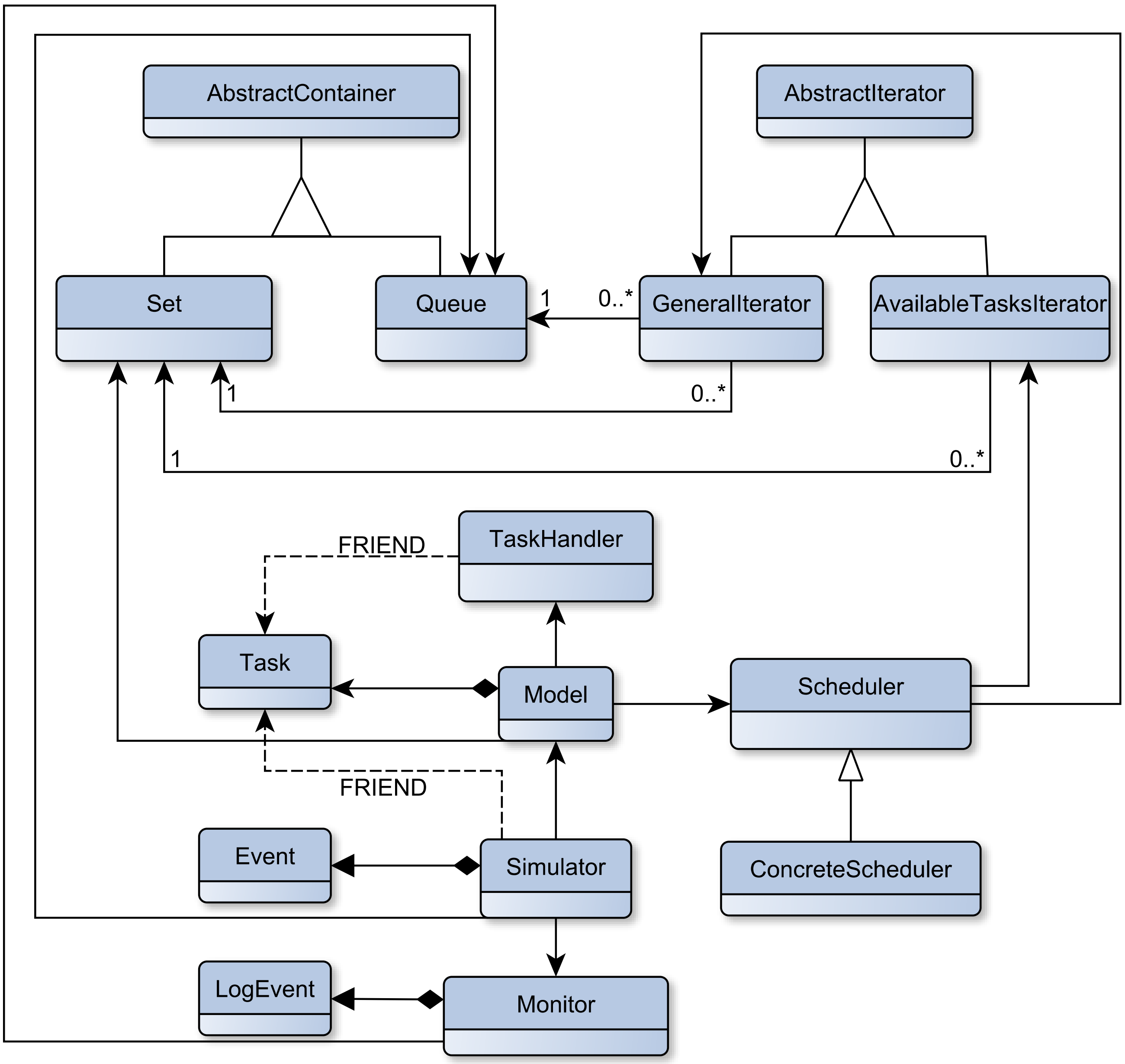


Figure 6.4: The Scheduler Simulator system simplified class diagram

Generated Results Report

After a simulation, a report is generated, containing relevant information about the simulation and the behavior of the simulated scheduler. The information included in this report is listed in table 6.1.

Table 6.1: Simulation Output Report

What is included?	**Why is it included?**
Task set and the tasks data members	To record the simulated task set.
Average Turnaround Time	To measure the efficiency of the simulated scheduler.
CPU Idle Time	To measure how well the scheduler exploit the available processing time.
Number of deadlines reached	The number of deadlines reached within the simulation time.
Number of deadlines missed	The number of deadlines missed within the simulation time.
The number of Completed tasks	The number of tasks completed at least one time within the simulation time.
Results in Percentages	Idle CPU time
	Deadlines missed
	Completed Tasks

6.2.2 Task Class

Friend classes: `TaskHandler`, `Simulator`.

The `Task` class represents operations or threads to be executed by the simulator. A `Task` object can have different states throughout the simulation. The states are listed in table 6.2.

Table 6.2: Task class States

`State`	**When is a `Task` in this state?**
`IDLE`	When a task is waiting for its Arrival Time, or a new cycle to begin (if task is periodic).
`READY`	When a task is ready to be executed.
`RUNNING`	When a task is being executed.
`WAITING`	When a task is interrupted by a higher prioritized task.
`BLOCKED`	When a task is blocked from executing because its waiting for a lower prioritized task to finish or unlock a certain variable.
`FINISHED`	When a task is finished executing.

Table 6.3: `Task` class data members

`Data Member`	Explanation.
`ExecutionTime`	The tasks Execution Time.
`Progression`	The execution progression of the task.
`Tarrival`	Arrival Time for the task.
`Deadline`	The relative deadline.
`Priority`	The priority set by the scheduler.
`Period`	The period of the task if periodic, otherwise zero.
`State`	The tasks current state.
`ID`	The task identifier.

The `Task` class contains data members which can be used to schedule and simulate an execution of the `Task` object. The `Task` data members can be seen in table 6.3. The complete `Task` class can be seen in figure 6.5.

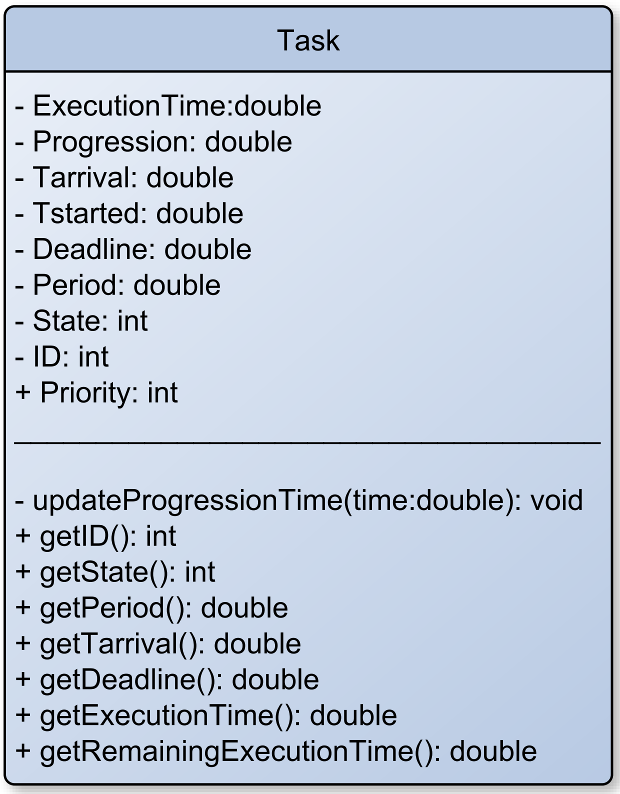


Figure 6.5: The Task class

6.2.3 Model Class

The `Model` class serves as a container for the parameters needed to conduct a simulation. The `Model` class data members can be seen in table 6.4.

Table 6.4: `Model` class data members

`Data Member`	**Explanation**
`ModelName`	A name for the model, used to name resulting reports.
`TaskSet`	A set of task pointers.
`Scheduler`	A `Scheduler` object containing the logic for a scheduling policy.
`RunTime`	The length of the simulation, default set to 150 `milliseconds`.
`ContextSwitch`	The context switch value, defaule set to 0 `milliseconds`.
`TimeInterval`	The clock cycle, default set to 1 milliseconds.
`TaskHandler`	A handler to generate and update task, check for deadline misses, and related functions.

The `Model` class has multiple constructors, giving the user alternatives on how to create a `Model` object and specify its values. The different `Model`-constructors are listed in table 6.5.

Table 6.5: `Model` class constructors

`Model()`	A default constructor. Default values.
`Model(modelName, modelTaskSet, modelScheduler, modelRunTime)`	Presets some data members using the arguments in the constructor.
`Model(modelName, numberOfPeriodicTasks, numberOfAperiodicTask, modelScheduler)`	Generates a number of periodic and aperiodic tasks based on the arguments in the constructor.

The `Model` class can be seen in figure 6.6.

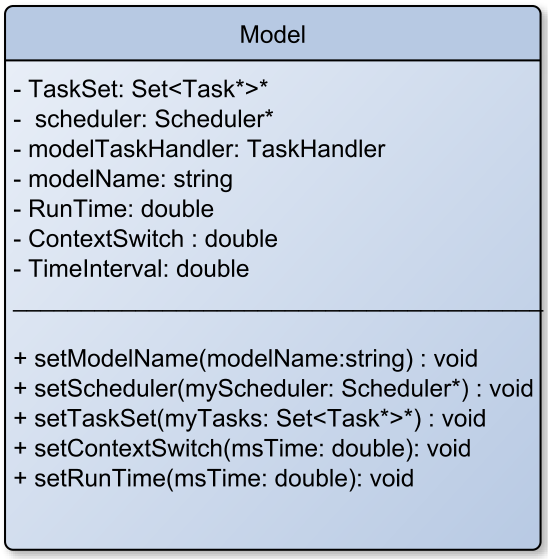

Model
- TaskSet: Set<Task*>*
- scheduler: Scheduler*
- modelTaskHandler: TaskHandler
- modelName: string
- RunTime: double
- ContextSwitch : double
- TimeInterval: double
+ setModelName(modelName:string) : void
+ setScheduler(myScheduler: Scheduler*) : void
+ setTaskSet(myTasks: Set<Task*>*) : void
+ setContextSwitch(msTime: double): void
+ setRunTime(msTime: double): void

Figure 6.6: The `Model` class

6.2.4 TaskHandler Class

The `TaskHandler` is a class used by the `Model` class to handle the task set. The `TaskHandler` class is a friend class of the `Task` class, thus it can change and access `Task` class data members. The main functionality of the `TaskHandler` is to generate task sets, update task states and check for deadline breaches. The `TaskHandler` class can be seen in figure 6.7.

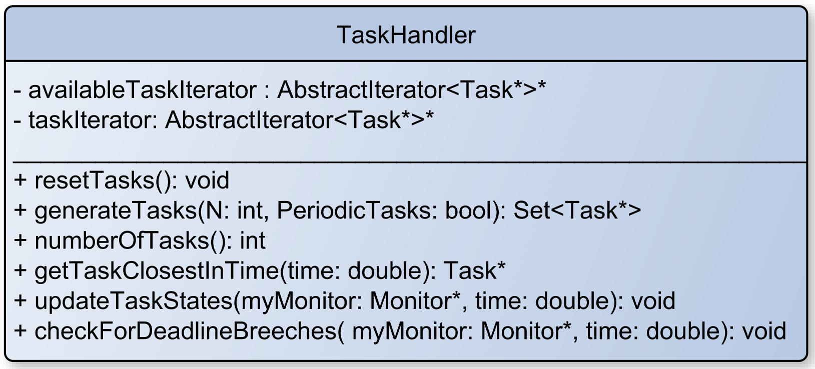

TaskHandler
- availableTaskIterator : AbstractIterator<Task*>*
- taskIterator: AbstractIterator<Task*>*
+ resetTasks(): void
+ generateTasks(N: int, PeriodicTasks: bool): Set<Task*>
+ numberOfTasks(): int
+ getTaskClosestInTime(time: double): Task*
+ updateTaskStates(myMonitor: Monitor*, time: double): void
+ checkForDeadlineBreeches(myMonitor: Monitor*, time: double): void

Figure 6.7: The `TaskHandler` class

6.2.5 Simulator Class

The `Simulator` conducts simulated executions of tasks. The task execution order is decided by the scheduling policy. The `Simulator` can be used to run simulations of `Model` objects. The `Simulator` contains an queue of `Event` pointers, and reacts to the `Event` closest in time. The `Simulator` class can be seen in figure 6.8.

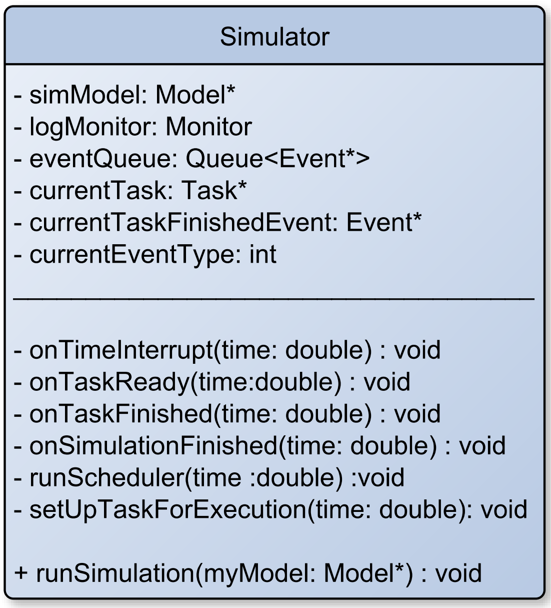

Simulator
- simModel: Model*
- logMonitor: Monitor
- eventQueue: Queue<Event*>
- currentTask: Task*
- currentTaskFinishedEvent: Event*
- currentEventType: int
- onTimeInterrupt(time: double) : void
- onTaskReady(time:double) : void
- onTaskFinished(time: double) : void
- onSimulationFinished(time: double) : void
- runScheduler(time :double) :void
- setUpTaskForExecution(time: double): void
+ runSimulation(myModel: Model*) : void

Figure 6.8: The `Simulator` class

6.2.6 Event Class

The `Event` class represents the possible events that can occur during a simulation. It contains a time stamp and an `EventType`. The different `EventTypes` are listed in table 6.6.

Table 6.6: EventTypes

EventType	**Explanation**
`TimeInterrupt`	Represents the clock-cycle. Pre-emptive schedulers can interact on TimeInterrupts.
`TaskFinished`	An Event indicating that the current task being executed is finished.
`TaskReady`	Indication that at least one Task object is ready for execution at this time.
`SimulationFinished`	Simulation run time is finished.

The `Event` class is shown in figure 6.9.

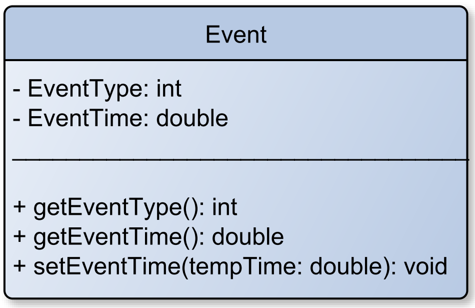
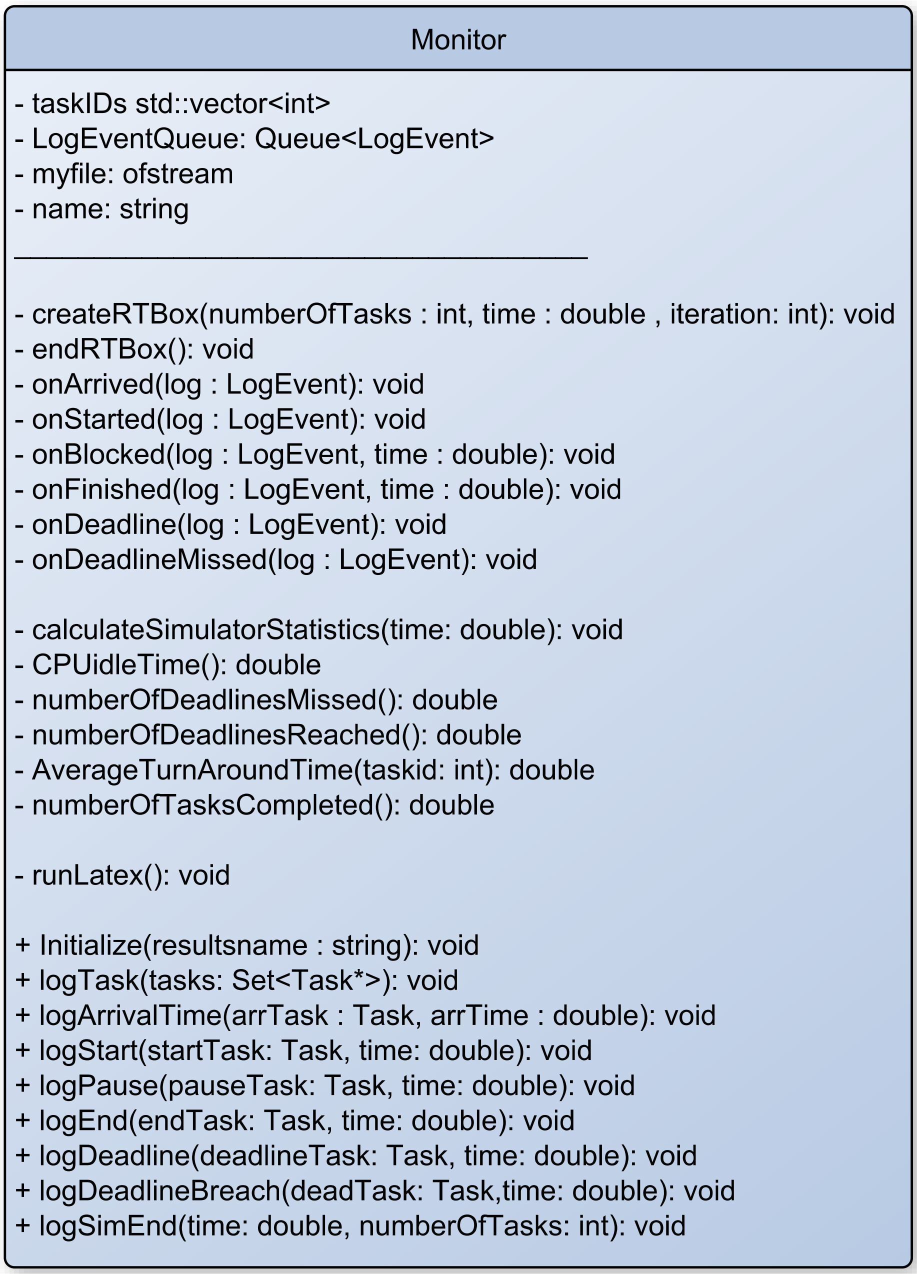


Figure 6.9: The `Event` class

6.2.7 Monitor Class

The `Monitor` class observes the simulation, and keep track of a queue of `LogEvents`. The `Monitor` generates relevant reports after a simulation has completed. The *"log"*-functions from the `Monitor` class is called upon by other classes to log incidents. The `Monitor` class can be seen in figure 6.10.



```
Monitor

- taskIDs std::vector<int>
- LogEventQueue: Queue<LogEvent>
- myfile: ofstream
- name: string
________________________________________

- createRTBox(numberOfTasks : int, time : double , iteration: int): void
- endRTBox(): void
- onArrived(log : LogEvent): void
- onStarted(log : LogEvent): void
- onBlocked(log : LogEvent, time : double): void
- onFinished(log : LogEvent, time : double): void
- onDeadline(log : LogEvent): void
- onDeadlineMissed(log : LogEvent): void

- calculateSimulatorStatistics(time: double): void
- CPUidleTime(): double
- numberOfDeadlinesMissed(): double
- numberOfDeadlinesReached(): double
- AverageTurnAroundTime(taskid: int): double
- numberOfTasksCompleted(): double

- runLatex(): void

+ Initialize(resultsname : string): void
+ logTask(tasks: Set<Task*>): void
+ logArrivalTime(arrTask : Task, arrTime : double): void
+ logStart(startTask: Task, time: double): void
+ logPause(pauseTask: Task, time: double): void
+ logEnd(endTask: Task, time: double): void
+ logDeadline(deadlineTask: Task, time: double): void
+ logDeadlineBreach(deadTask: Task,time: double): void
+ logSimEnd(time: double, numberOfTasks: int): void
```

Figure 6.10: The `Monitor` class

6.2.8 LogEvent Class

The `LogEvent` class represents the different types of events that can occur during simulation that should be logged in order to generate relevant reports and statistics. The different types of `LogEvents` are listed in table 6.7.

Table 6.7: LogEvent types

`LogEventType`	**When is the `LogEvent` created?**
`logARRIVED`	When a task has reached its arrival time.
`logSTARTED`	When a task has started executing.
`logBLOCKED`	When a task is blocked from executing due to another task.
`logFINISHED`	When a task finishes executing.
`logDEADLINE`	The deadline time for a task.
`logDEADLINEMISSED`	When a task misses its deadline.

The `LogEvent` class is shown in figure 6.11.

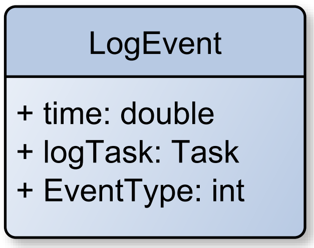


Figure 6.11: The `LogEvent` class

6.2.9 Scheduler Class

The `Scheduler` class represents the simulated scheduler. A `Scheduler`-pointer is used by the `Model` class, connecting the `Scheduler` class to a task set. The `Scheduler` contains a pure virtual function, so in order to conduct a simulation, a sub-class of the `Scheduler` class must implement this function. The `Schedulers` data members are listed in table 6.8.

Table 6.8: `Scheduler` Data Members

`Scheduler` **Data Members**	**Explanation**
`availableTaskIterator`	Iterator that only points to Task objects ready to be executed.
`TaskIterator`	Iterator that traverse the entire task set.
`preemptive`	Indication if the scheduler is preemptive or not in behavior.

The `Scheduler` class can be seen in figure 6.12.

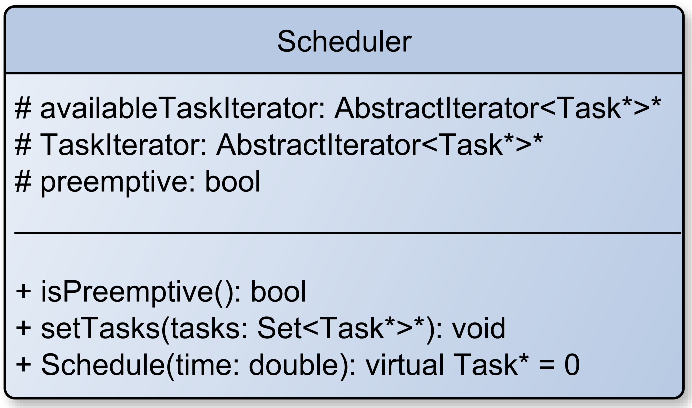

Scheduler
# availableTaskIterator: AbstractIterator<Task*>*
# TaskIterator: AbstractIterator<Task*>*
# preemptive: bool
+ isPreemptive(): bool
+ setTasks(tasks: Set<Task*>*): void
+ Schedule(time: double): virtual Task* = 0

Figure 6.12: The `Scheduler` class

6.2.10 Containers and Iterators

To create sets and queues of objects, containers with customized iterators are used. A simplified class diagram for this system is shown in figure 6.13.

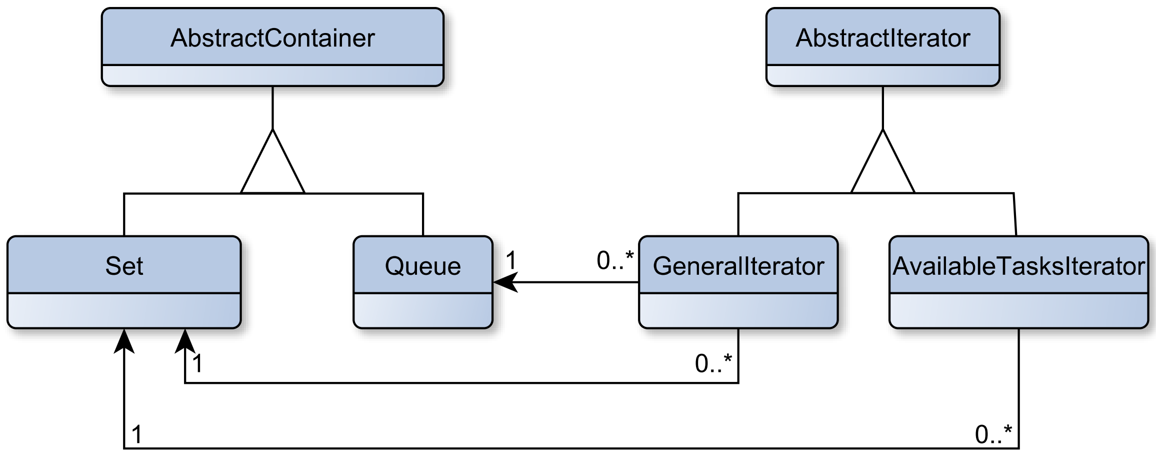


Figure 6.13: The simplified Container and Iterator system class diagram

Container classes

The `AbstractContainer` is an abstract class, defining the interface for interaction with the containers. Two sub-classes exists:

- The `Set` class stores its objects in an std::vector<Item>. This class is intended for storing Task objects.

- The `Queue` class stores objects in a std::list<Item>. This class is intended for storing Event objects and LogEvent objects.

The Container class diagram is shown in figure 6.14.

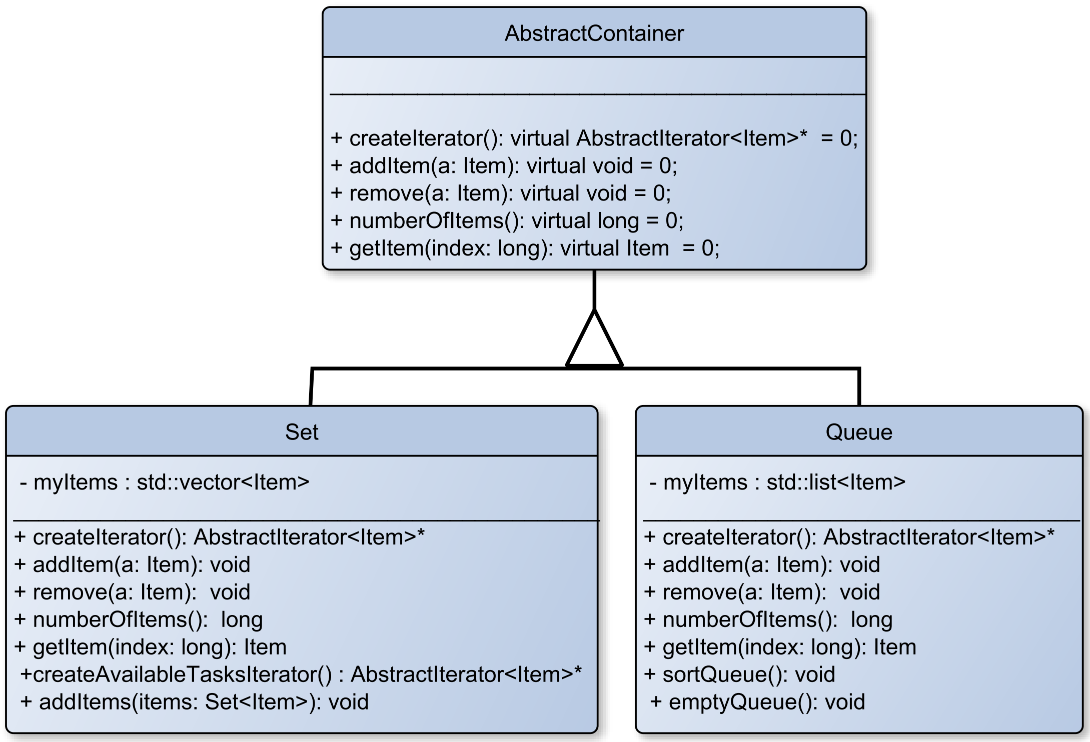

AbstractContainer
+ createIterator(): virtual AbstractIterator<Item>* = 0;
+ addItem(a: Item): virtual void = 0;
+ remove(a: Item): virtual void = 0;
+ numberOfItems(): virtual long = 0;
+ getItem(index: long): virtual Item = 0;
Set
- myItems : std::vector<Item>
+ createIterator(): AbstractIterator<Item>*
+ addItem(a: Item): void
+ remove(a: Item): void
+ numberOfItems(): long
+ getItem(index: long): Item
+createAvailableTasksIterator() : AbstractIterator<Item>*
+ addItems(items: Set<Item>): void
Queue
- myItems : std::list<Item>
+ createIterator(): AbstractIterator<Item>*
+ addItem(a: Item): void
+ remove(a: Item): void
+ numberOfItems(): long
+ getItem(index: long): Item
+ sortQueue(): void
+ emptyQueue(): void

Figure 6.14: The Containers class diagram

Iterator classes

The `AbstractIterator` is an abstract class, defining the interface for traversing an `AbstractContainer`. Two sub-classes exists:

- The `GeneralIterator` can be used to traverse the entire content of both the `Set` container and the `Queue` container.

- The `AbailableTaskIterator` is intended for traversing a `Set` container with Task- objects, iterating only through Tasks that can be executed.

The Iterator class diagram is shown in figure 6.15.

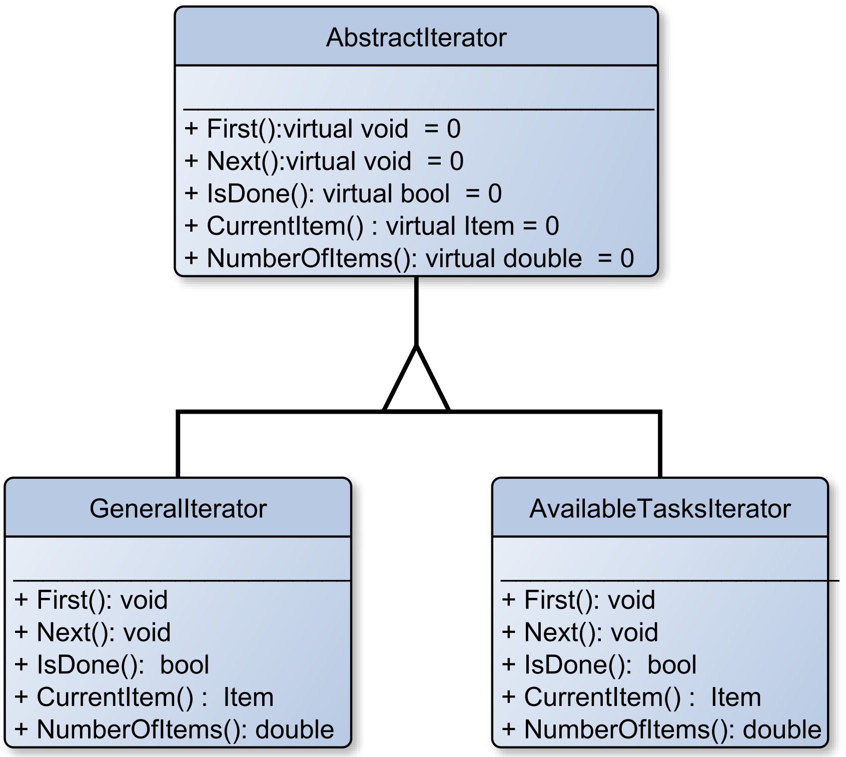

AbstractIterator
+ First():virtual void = 0
+ Next():virtual void = 0
+ IsDone(): virtual bool = 0
+ CurrentItem() : virtual Item = 0
+ NumberOfItems(): virtual double = 0
GeneralIterator
+ First(): void
+ Next(): void
+ IsDone(): bool
+ CurrentItem() : Item
+ NumberOfItems(): double
AvailableTasksIterator
+ First(): void
+ Next(): void
+ IsDone(): bool
+ CurrentItem() : Item
+ NumberOfItems(): double

Figure 6.15: The Iterator class diagram

6.3 User Documentation: Final Documentation

To utilize, maintain and improve the Scheduler Simulator, the user must understand the behavior as well as the implementation of the software.

6.3.1 Navigating the Scheduler Simulator Directory

The ***SchedulerSimulator/*** **directory** consists of several folders:

The *top level* contains:

- the Scheduler class.
- the SchedulerSimulator source file.

The *SimulatorCore* folder contains:

- The Task class.
- The Simulator class.
- The Event class.
- The Model class.
- The TaskHandler class.

The *Monitor* folder contains:

- The Monitor class.
- The LogEvent class.

The *Schedulers* class contains the implemented scheduling policies.

The *Container* folder contains the container classes.

The *Iterator* folder contains the iterator classes.

The *Results* folder will contain generated result reports.

6.3.2 System Behavior

The purpose of the system is described in the **requirement documentation** in section 6.1.
The main behavior of the system is located in the `Simulator` class. The event-driven behavior is illustrated by a flow chart in figure 6.16.

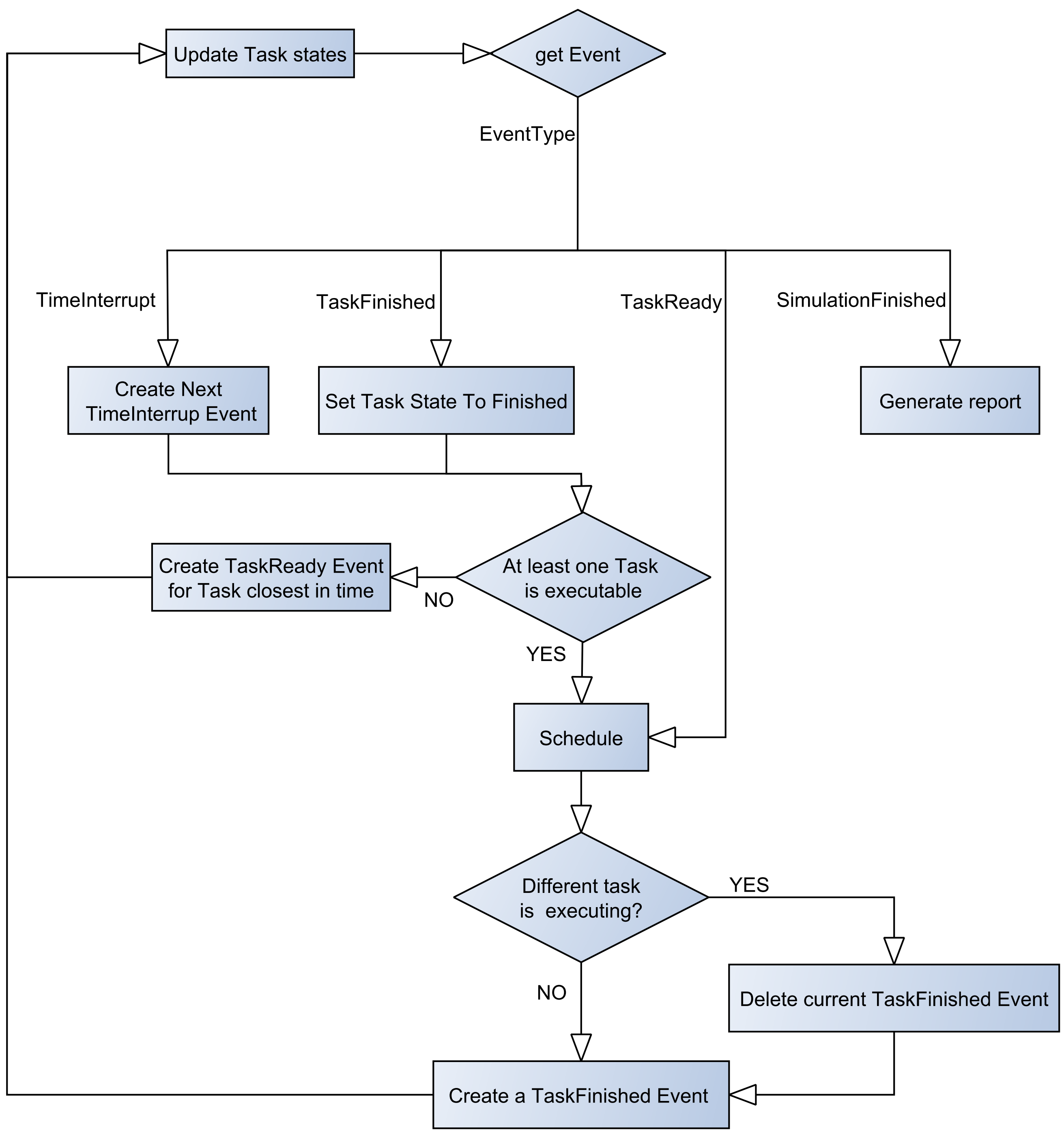

Update Task states
get Event
EventType
TimeInterrupt
TaskFinished
TaskReady
SimulationFinished
Create Next TimeInterrup Event
Set Task State To Finished
Generate report
Create TaskReady Event for Task closest in time
NO
At least one Task is executable
YES
Schedule
Different task is executing?
YES
NO
Delete current TaskFinished Event
Create a TaskFinished Event

Figure 6.16: A flow chart illustrating the behavior of the simulator.

6.3.3 System Implementation

The system is implemented in C++, using non-restricted libraries to maintain the cross-platform functionality. Some of the functions used in the software requires a C++11 compiler. The Scheduler Simulator comes with both a Makefile for Unix platforms, and a Microsoft Visual Studio Solution for Windows platforms.

Descriptions of classes and class diagrams can be found in the design documentation, while actual header files displays the actual implemented versions.

System design and architecture documentation can be found in section 6.2.
Code documentation: see header files in appendix B.

6.3.4 System Limitations

There are some limitations currently associated with the `Scheduler Simulator`.

`Task`-objects contained in the same Set must have unique ID values. Generated task sets will automatically have unique identifications, but there is no safety-guard currently implemented that ensures that the `Task`-objects in manually implemented task sets have unique ID values.

If `Model` objects contain the same model name, and are run sequentially, the resulting reports will be overwritten. Unique `Model` names should be provided to avoid loss of results.

6.4 User Documentation: End-User Documentation

This documentation will enable the reader to use the features of the `Scheduler Simulator`.

The `Scheduler Simulator` is a scheduling policy simulator written in C++. The simulator can create task sets and simulate them with various scheduling policies. It is also possible to create and add new scheduling policies.

6.4.1 Get Started

To generate reports, a proper installation of LaTeXis needed. The Scheduling Simulator can be used on both Windows and Unix platforms.
On Windows: Open the provided Microsoft Visual Studio Solution and run the project. Other compilers can be also used, using the source files for the Scheduler Simulator.
On Unix: Enter the `"SchedulerSimulator/"` directory and run a `make` command in the terminal, followed by a `"./Simulator"` command to run the simulation.

The user must have some experience in C++ programming language. The user also needs to familiarize him/her self with a small part of the program in order to use the software correctly.

The following two **tutorials** enables the user to utilize the Scheduler Simulator. The tutorials describe the procedures to **simulate a scheduler**, section 6.4.2, and to **implement a new scheduling policy** in section 6.4.3.

6.4.2 How to Simulate a Schedulers Behavior

To simulate a schedulers behavior, the user must

- Declare which scheduler to simulate
- Generate or declare a task set
- Create a model containing the scheduler and the task set
- Simulate the model.

A set-up for doing this is provided in the source file SchedulerSimulator.cpp, as shown below. For more details, read the following sections.

```
int main()
{
    //Declare the schedulers
    FIFO myScheduler;

    //For Manually creating tasks:
    //Create a Task Set:
    Task a(1, 0, 100, 24, 100);
    Task b(2, 0, 80, 20, 80);
    Task c(3, 0, 60, 20, 60);

    Set<Task*> myTasks;
    myTasks.addItem(&a);
    myTasks.addItem(&b);
    myTasks.addItem(&c);

    //Create a model
    Model myModel("MyModel", &myTasks, &myScheduler, 150);


    //For generating tasks within the model:
    //Create a model with 3 periodic and 4 aperiodic tasks
    Model generatedTasksModel("generatedTasks", 3, 4, &myScheduler);
    //Set runtime
    generatedTasksModel.setRunTime(200);

    //Create a simulator
    Simulator mySimulator;

    //Run simulations

    mySimulator.runSimulation(&myModel);
    mySimulator.runSimulation(&generatedTasksModel);
}
```

Declare a Scheduler to Simulate

Declaring a scheduling policy is done by creating the corresponding scheduling object.

```
schedulingObject myScheduler
```

If wanting to simulate the FIFO scheduling policy, the scheduler is declared as below:

```
    FIFO myScheduler;
```

Four scheduling policies are implemented, available for simulation:

- RR
- RM
- EDF
- FIFO

Creating a Task set

A simulation needs a task set to schedule. Creating a task set can happen in two ways: manual declaration, or generated by the Model.

Manual declaration:

1. Create tasks using the constructor: `Task(int id, double tarrival, double deadline, double executiontime, double period);`
2. Create a Set with task pointers: `Set<Task*> myTasks;`.
3. Add the Tasks to the Task set: `myTasks.addItem(& myTask);`

Remember: provide tasks individual IDs. Example code in C++ can be seen below.

```
//1. Create tasks
    Task a(1, 0, 100, 24, 100);
    Task b(2,0, 80,20,80);
    Task c(3, 0, 60, 20, 60);

//2.Create a Set with task pointers
```

```
    Set<Task*> myTasks;

//3.Add the Tasks to the Task set
    myTasks.addItem(&a);
    myTasks.addItem(&b);
    myTasks.addItem(&c);
```

Generate Task Set:

1. When creating a model, use the constructor to provide how many periodic and aperiodic tasks to simulate.

The generated task set will be created using normal distribution. Example code in C++ can be seen in the section below on how to **create a model**.

Create a Model

A model combines a scheduler with a task set, and provides the needed parameters and information for the simulator. There are several options on how to create a model. The modelName will be used as a file name for the resulting documentation of the simulation.

1. Create the model using one of three constructors:

   (a) `Model():`
   sets default values to the parameters. Task Set must be added later.

   (b) `Model(std::string modelName, Set<Task*>* modelTaskSet, Scheduler* modelScheduler, double modelRunTime):`
   Creates a model with the specified arguments.

   (c) `Model(std::string modelName, int numberOfPeriodicTasks, int numberOfAperiodicTask, Scheduler* modelScheduler):`
   Creates a model with a generated task set.

2. Specify parameters if needed.

Example code showing the different ways of creating a model is shown below:

```
// Creating a model using the default constructor:
    Model myModel;
    myModel.setModelName("TestName");
    myModel.setScheduler(&myScheduler);
    myModel.setTaskSet(myTasks);
    myModel.setRunTime(200);

// Creating a model using the specified constructor:
    Model myModel("TestName", myTasks, &myScheduler, 200);

// Creating a model to generate a task set with
// 3 periodic tasks and 4 aperiodic tasks:
    Model myModel("TEST", 3, 4, &myScheduler);
```

Simulate a Model

To simulate the model, a Simulator object is needed. A Simulator can simulate multiple models.

1. Create a Simulator object.

2. Run simulation of the model.

Example code is provided below:

```
//  1. Create a Simulator object:
    Simulator mySimulator;

//  2. Run simulation of the model:
    mySimulator.runSimulation(&myModel);
```

Results

The resulting documentation from the simulation can be found in the `.../SchedulerSimulator/Results/` folder if not opened automatically. The file name is the modelName from the simulated model. The documentation provides relevant information about the simulation, resulting statistics and charts.

6.4.3 Implement a Scheduler

To add a new scheduling policy for simulation, the following procedure must be executed:

- Create a class inheriting the Scheduler class.
- State in the constructor if the scheduling policy is **preemptive** or not, using the `preemptive` boolean inherited from the base class.
- Implement the scheduling logic in the virtual inherited function `Task* Schedule(double time)`.
- Include the class header in `SchedulerSimulator.cpp`.

For more details, please read the following sections.

Create a New Scheduler Class

The new scheduler class must inherit the base Scheduler class. It must be stated in the new scheduling class' constructor whether the scheduling policy is preemptive or not. If preemptive, the scheduler is allowed to interrupt tasks currently being executed in order to execute a different task. If not preemptive, a task must finish executing before another task can start executing. An example on how to create a new scheduler can be seen below:

```
#pragma once
#include "../Scheduler.h"

class EDF :
    public Scheduler
{
public:
    EDF();
    ~EDF();
    Task* Schedule(double time);
};

//constructor:
```

```
EDF::EDF()
{
    preemptive =true;
}
```

Implementing the Scheduling Logic

The new scheduler has to implement the virtual function `Task* Schedule(double time)`, where the time corresponds to the progress of the simulation, and the `Task*` is a pointer to the task the scheduler wants to admit to the simulator for execution. There are two ways to traverse the task set: the **TaskIterator** and the **availableTaskIterator**.
The TaskIterator iterates through the entire task set. *The availableTaskIterator* iterates through the tasks that are ready to execute, including the task currently being executed by the simulator. The scheduler can only return a task pointed to by the `availableTaskIterator`, but the `TaskIterator` enables the scheduler to access all tasks in the task set. The Iterators are both subclasses of the base class AbstractIterator, thus both can be interacted with using the interface shown in figure 6.17.

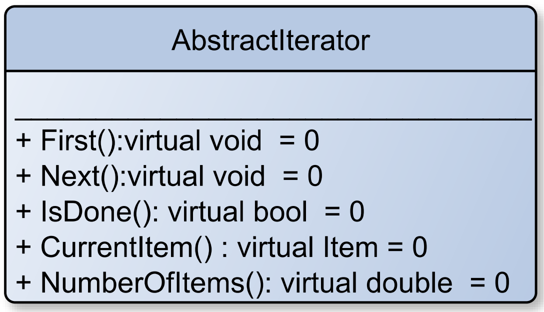

AbstractIterator
+ First():virtual void = 0
+ Next():virtual void = 0
+ IsDone(): virtual bool = 0
+ CurrentItem() : virtual Item = 0
+ NumberOfItems(): virtual double = 0

Figure 6.17: The AbstractIterator interface.

To reach a scheduling decision, information about the tasks must be gathered and compared. This is done by using the task interface, shown in figure 6.18. The only writable parameter is `priority`, because this is only relevant for the scheduler.

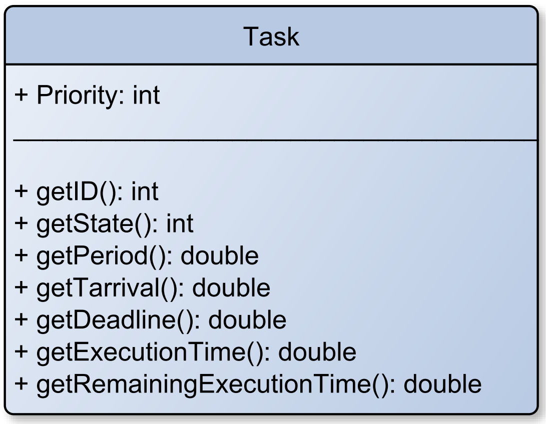


Figure 6.18: The task interface relevant for the scheduler.

Using these interfaces, the logic is implemented in `Task* Schedule(double time)`. An example of how it can be done for the Earliest Deadline First is shown below:

```
Task* EDF::Schedule(double time)
{
    double tempAbsDeadline = 0.0;
    double tempTarrival = 0.0;
    double tempRelativeDeadline = 0.0;

    availableTaskIterator->First();
    Task* bestTask = availableTaskIterator->CurrentItem();
    double closestDeadline = bestTask->getDeadline();

    for (availableTaskIterator->First(); !availableTaskIterator->IsDone();
                                            availableTaskIterator->Next())
    {
        tempTarrival = availableTaskIterator->CurrentItem()->getTarrival();
        tempRelativeDeadline = availableTaskIterator->CurrentItem()->getDeadline();

        tempAbsDeadline = (tempTarrival+ tempRelativeDeadline);

        if (tempAbsDeadline < closestDeadline)
        {
            closestDeadline = tempAbsDeadline;
            bestTask = availableTaskIterator->CurrentItem();
        }
    }
    return bestTask;
}
```

Place the header and source file in the `/SchedulerSimulator/Schedulers/` folder preferably, and include the corresponding header file in `SchedulerSimulator.cpp`. Continuing the EDF example, this is done using: `#include"/schedulers/EDF.h"`. The scheduler is now ready to be simulated.

Chapter 7

Discussion

For this project, the created `Simulator Scheduler` software, the produced documentation and the development process itself are all considered to be project results.

7.1 Scheduler Simulator Test Evaluation

The test run results are displayed in chapter 5.

7.1.1 Test with Manually Created Task Set

The manually created task set was simulated with all four pre-implemented scheduling policies. The results can be seen in section 5.1. Based on the descriptions of the scheduling policies in section 3.1.3, the resulting Gantt charts correspond with the expected scheduling behavior for all four simulated schedulers. All the simulations resulted in 6% CPU idle time, but only the EDF scheduler reached all the deadlines. In the Gantt chart for the FIFO simulation, section 5.1.2, a deadline is missed at time 60. The read arrow seems to be partially hidden behind the tab marking the tasks execution. This is assumed to be caused by of the order of when the different elements were generated in the chart. The deadline miss is visible, but not as clearly marked as it was intended to be.

7.1.2 Test with Generated Task Set

Two generated task sets were simulated using the EDF scheduling policy. The resulting reports from the simulations can be seen in section 5.2. All the tasks in the two task sets have generated reasonable values, and are executed based on these values. In the simulation of the task set containing 20 tasks, the Gantt chart almost overshoots the page of the generated pdf file, see section 5.2.2. Simulating more than 20 tasks will thus make the Gantt chart so large that some tasks might end up outside of the page, leading to an incomplete chart. Nevertheless, the resulting statistics and information for simulations of large task sets can still be produced and displayed in the report. Also here, in the Gantt diagram for the simulation of twenty tasks, seen in section 5.2.2, a deadline is poorly marked at time 90 as it was for the FIFO-simulation, see section 7.1.1.

7.2 User Interface Evaluation

The main features offered to the user by the created software are:

- Running a simulation of a real-time scheduler
- Implementing a new real-time scheduler

Following the user tutorials provided in section 6.4, the user should be able to conduct these actions without much trouble. *To run a simulation*, the user has to make changes in the `main` function of the software. A user can in theory write anything in the `main` function. If the user does not understand the provided documentation, no strict rules constraint the user to follow the developers intentions. A *GUI* can possibly provide the user with a more strict and limited interface, inducing a safer work environment to work in. *The "Facade"* pattern, section 2.3.2, implemented correctly, could have provided the user with a simple way to find relevant functions and conduct a simulation in main, without the use of a GUI. Both these solutions would have shrunk the freedom of the user, inducing both positive and negative side effects. The negative being limited expansion and exploration possibilities for the user. A software can survive and evolve in the hands of curious users. The current text-based solution, though unrestricted, might be the best way to create a platform for future expansion of the software.

7.3 Simulating Future Schedulers

The Scheduler Simulator enables the user to add schedulers by implementing a new class. Done correctly, the user will be enabled to run simulations using the newly implemented scheduler. To ensure that the scheduler behaves in the intended manner, the user has to examine the resulting simulation report.
The `Scheduler Simulator` is meant to accommodate all schedulers, existing and future types. As there is no way to know how the schedulers in the future might behave, the implementation of the schedulers are created based on the one common fact for all schedulers: the have to be able to *schedule.* The `Scheduler` base class is only

expecting its sub-classes to implement one function, the `"Task* schedule(double time)"` function. Any algorithm for a current or future scheduler can be implemented here. *Possible restrictions* limiting the implementation freedom might be the can be lack of depth and expansions in other parts of the software. For example, the data members of the `Task` class may not contain all the information a future scheduling policy might need. In such cases, the software must be expanded to accommodate unforeseen new scheduling policies. This way, the software becomes more and more adapted to simulate future schedulers.

A different solution could be to let the *Scheduler* base class contain more than one virtual functions, splitting the scheduling procedure in mandatory steps. This might create a safer and easier implementation procedure for existing schedulers, but it would create constraints on the schedulers behavior, possibly limiting the ability to implement future scheduling policies.

7.4 Documentation Evaluation

Creating light, yet sufficient documentation can be a challenging task. The complexity of the software must be communicated to the reader without making the explanations overly complicated. The documentation for the Scheduler Simulator, created in this project, can be seen in chapter 6.

The requirement documentation, seen in section 6.1, used written text descriptions, user stories and three use-cases to present the requirements for the software. All these methods are list-based, and no narrative explanations were provided. This provides an ordered and straightforward presentation of the requirements, but a narrative text could have helped some readers to grasp the concept of the software feature, though for others it would be redundant. One of the use-cases represents the simulator as the actor, and the scheduler as the system. This is not the "classical" way of creating use-cases, as the simulator is not an external system actor, but actually part of the system itself. Even so, this use-case was created and kept because it serves the purpose of presenting the wanted simulator behavior in an ordered, readable manner.

The architectural documentation in section 6.2 presented the classes and their interfaces in a strict and efficient way using diagrams. The short explanations may be at the expense of descriptive and pedagogical text that would have made it easier for some readers to understand the architectural design of the software. Long and narrative explanations may cause other readers to skip paragraphs, because they just need key points, and are not interested in reading a wall of text to extract the information they really want. The final architectural documentation can be used as a quick class reference manual by users or maintenance personnel.

The user documentation consists of two parts:

> *The final documentation* is a short to-the-point guide for users seeking an in dept explanation of the `Scheduler Simulator`. The readers for this section are presumed to be familiar with C++ language, and so the purpose of this documentation was to provide the user with the necessary tools to gain understanding of the systems structure and behavior. A clear point list of features, a class reference manual, a flow chart of the simulators behavior, and a commented source code might be more beneficial for an experienced user than a long and narrative explanation.
>
> *The end-user documentation* contains two tutorials on how to *run a simulation* and how to *implement a scheduler*. The tutorials are text-based, combining explanatory text, code examples and class diagrams to show the user step by step how to use the features of the `Scheduler Simulator`. The tutorials provide the needed information for a basic end-user, no more or less. Hopefully, a user may use these tutorials as a check list when conducting new simulations or when implementing new scheduling policies.

Light and sufficient documentation is the goal many developers are aiming for, but in many cases the choice seems to become "light *or* sufficient documentation". In this project, long and complicated texts were actively avoided, but the finished documentation is still over 25 pages. The length is necessary to accommodate the different needs of the variety of users. A normal user would not have to read more than the end-user documentation, and utilize the rest of the documentation as a reference manual.

7.5 Evaluation of Best Practices

The development of the `Scheduler Simulator` was conducted using a set of elements from relevant best practices. Table 4.1 provides an overview of the utilized best practices within software development methods.

Agile development, being a collective methodology for light weight methods, emphasizes project flexibility to meet the client's requirements. Arranging frequent meetings with the client, presenting prototypes and making sure the requirements and progress corresponds to the clients wishes helps to achieve this. In this project, the authors supervisor was the client. The author had frequent meetings with the supervisor for counseling, but only one strictly client- developer meeting was held to discuss the specification for the software early in the process, see report in A.1. This vague separation between the supervisors roles of being both a counselor and a client, led to few strict client-developer meetings. Event so, the developer still had frequent contact with the client indirectly through the counseling meetings, presenting iterative results and receiving positive feedback. In Agile Methodology, the point of the frequent contact with the client, is ensure the client and the developer that they are on the same page, keeping the developer from heading down the wrong path when developing. Even though few strict client-developer meetings were held, the counseling meetings provided much of the same acknowledgement and feedback as potential direct client meetings would have provided. The council meetings let the developer know that the ongoing development and results corresponded with the supervisors wishes.

Nevertheless, in a different situation, a client would most likely have less experience with software development than the authors supervisor, making direct client-developer meetings with presentations of prototypes and progress during the development important and more challenging. In this case, a larger use of use-cases and user stories might have been helpful to make sure that the developer understood the clients wishes correctly. The supervisors understanding of software systems and knowledge of technical terminology made the client- developer conversations flow easily, requirements were discussed as equals, and misunderstandings were more easily avoided.

The daily Scrum was intended to be used for this development project, but it was not documented, nor was it completed every day. Being only one person, the daily Scrum meetings became more of a frequent thought process, a reminder of what remained to be done on the project. If a stricter routine had been set, more reflection and possibly productivity could have spawned from the daily Scrum. The author could have, for example, created a log, demanding her self to make daily entries answering the daily scrum questions.

Iterative and incremental development was utilized during the development project. It helped to create the finished product by gradually expanding the specification for the software through iterative cycles. Some iterations and features were developed and improved during a long period of time, and because the duration of the iterations were set using requirements goals, this was allowed to happen. In retrospect, spending too much time on one feature prevented the overall progress of the software. Using time-boxed sprints instead of requirement goals to define the length of an iteration could have averted such situations.

Keeping it simple when creating a software is a challenge. A simple idea for a software design can easily escalate to something much more complicated, as demonstrated in section 4.2.3. Simplicity is still something worth striving for, as it promotes good design and understandable code. It is not always achieved, but it helps the developer to at least start with a clean and simple structure, which prevents the software from rotting from the inside out.

The overall development progress had a tendency to converge towards a sequential *waterfall* structure. This can be caused by the fact that only one person was working on the software project, and a sequential line of working comes natural. In a larger group, parallel working increases productivity and progression, and enables independence and developing freedom between groups. As a single person, independence and freedom are already present, and parallel working can't invoke an increase in productivity, as one person only can type and work at one thing at a time.

Some of the *design patterns* described in section 2.3 were used in the system helped create maintainable code and robust software. New schedulers can be added to the system without altering any of the other classes thanks to the "Stratedy" pattern, and containers and iterators have general common interfaces because of the "Iterator" pattern. A greater use of patterns might have provided even cleaner and more reusable code, but the considering the current size of the software program, applying a pattern to cover a single occurrence of a class might serve against its purpose, by complicating and not facilitating the structure of the software. Design patterns could have been used to implement the event-based behavior in the simulator. The *observer* pattern would have created a one-to-many relationship where the simulator could have changed its state as a reaction to an event, and notified all dependent classes. The *state* pattern could have been used, creating one class per state to handle the current event. Both of these patterns would have created an increase of sub-classes in the system. As mentioned in section 7.2, the "Facade" pattern could have been used to create a clean and collected text-based user interface. Using known design patterns in a software enables others to quickly understand the implemented code if they already are familiar with the particular pattern.

7.6 Improvements and Shortcomings

The finished `Scheduler Simulator` is a working software, but shortcomings and room for improvement and expansions do exist.

The generated Gantt chart overshot the page size of the report when simulating task sets exceeding 20 tasks, see section 7.1.2. To accommodate larger task sets, Gantt diagrams for task sets exceeding 20 tasks are left ungenerated, as the resulting Gantt charts would be incompletely displayed. All the other simulation statistics and results will still be fully displayed.

As discovered in the test-runs, the deadline misses are not always clearly marked. The Gantt chart generation in the `rtsched` LaTeXstyle should be altered to include a bigger notification sign upon a deadline breach.

Each `Task` object in a task set must have a unique identification number. The current version of the simulator does not ensure this. A future version should include a safe guard for this problem. A possible solution is to let the `Set`-object containing the `Task` objects distribute unique ID values.

The resulting report can be expanded to include more statistics, such as *task response time.*

The implementation of the `Scheduler Simulator` is functional and sufficient for the current expected behavior. The implementation can still be improved to be more efficient, by improving the class structures and by optimizing algorithms with shorter estimated run-times. A software can almost always be optimized and improved, but working code must not be underestimated.

The goal of a simulator is to simulate a real life situation by emulating the actual circumstances. Adding features to the `Scheduler Simulator` can help the simulation to become more realistic. Such features are listed below.

- *Sporadic tasks* should be added to help accommodate all possible task types.
- When generating a task set, the user should be able to choose the *distribution* of the tasks. The current generator can only use normal distribution. This will expand the variations in task sets possible to generate.
- *Task dependencies* and shared variables should be implemented. This can

possibly be done by adding some sort of *"used-variable array"* to each `Task` object.

- *Jobs* are a set of tasks to be executed sequentially, see section 3.1.2. Jobs should be added to the software, and the simulator would execute a set of jobs instead of tasks. The main difference would be that only the one tasks in each job would be available for execution at a time.
- *Estimation of execution time* should be a feature done automatically for every task based on its properties.
- A feature to automatically *compare two or more schedulers* after a simulation should be included in a future version of the `Scheduler Simulator`.

Chapter 8

Conclusion

In this project, a software to simulate real-time schedulers was created, called the `Scheduler Simulator`. The simulator can successfully conduct simulations for the implemented schedulers: FIFO, EDF, RR and RM. Other schedulers can be implemented and added to the system. Because of the size, Gantt charts are not generated for simulations of task sets with more than 20 tasks. A GUI was evaluated to possibly be redundant, as the user must write source code when implementing new schedulers anyway. To enable simulations of future scheduling policies, the interface to implement new schedulers was created to be as unlimited as possible, containing only one mandatory function to implement. Possible limitations for implementing future schedulers can be caused by a lack of data parameters in the system, needed by the future scheduler's logic. The documentation for the `Scheduler Simulator` aims to present the relevant data in a structured way, using mainly tables, lists and diagrams, as opposed to long and narrative explanations. The documentation would rather highlight key points than to hide them in a explanatory text. This enables the user to utilize the documentation as a reference manual.

The software was developed using a set of best practices,. For the development, the author's supervisor portrayed the *client*. This lead to a vague distinction between counseling and development meetings, as the author did not need the clients approval of a prototype if the supervisor already had given positive feedback during a counseling meeting.
Agile development methodology emphasizes frequent contact with the client and project flexibility. *Iterative and incremental* development can increase flexibility in a development project, regardless of how many or few people that are working on it. In this project, iterative and incremental cycles bound by requirement goals were used to create the software. Some iterations were longer that others, due to the difference in time required to develop the different features. *Time-boxed*

Scrum sprints would have prevented this uneven development progression. Agile development methods promotes team work and uses social interaction to increase the flexibility in projects. When working alone, the social aspect ceases to exist, leading to a loss of the benefits that can be harvested by working in teams, such as pair-programming and social discussions to develop design, interfaces, implementation etc. When working on a project, work tendencies from the *waterfall model* may appear, as a natural sequential flow of work may seem logical and more appealing than iterative cycles. This is understandable, but should be avoided.

Creating a new software can include stepping in to unknown territory to discover new solutions. This may create a "the chicken or the egg" scenario, where knowledge is needed to advance to the next step, but this knowledge is also gained by finding the solution to advance to the next step. One might also find that by gaining knowledge, a new and better solution for an already solved problem might present itself. In inflexible development projects, these new and better solutions have little or no opportunity to be implemented in the software. Agile development methodology addresses and inspires developers to find these exact solutions and utilize them. New ideas can be found through development. Preserving and using these ideas is key for technological advancement.

8.1 Recommendations for Further Work

The Scheduler Simulator created in this project can be expanded in several ways to improve the authenticity of simulation. The `Task` class can be expanded to include *sporadic behavior, dependencies* and actual *estimations of execution time. Jobs* can be added, making the simulator simulate a set of jobs instead of individual tasks. The current software produces relevant reports per executed simulation and leaves it to the user to examine these reports in order to compare scheduler performances. A tool that compares scheduler performances and presents the results for the user can be useful, and should be implemented in the future.

Expansions to accommodate implementations of future schedulers when needed will help the `Scheduler Simulator` to grow and adapt to simulate all real-time schedulers in a realistic environment, changing the question in research on scheduling in real-time systems from *"How efficient can we be"* to *"How efficient do we want to be"*.

Bibliography

Ambler, S. (2001). *The Object Primer, Agile Model Driven Development with UML 2.* Cambridge University Press, third edition.

Balaji, M. (2012). Waterfall vs v-model vs agile: A comparative study on sdlc.

Barnett, G. (2008). Cpu scheduling simulator. Available at `http://cpuss.codeplex.com/`.

Beck, K. (2005). *Extreme Programming Explained.* John Wait, second edition.

Booch, Jacobson, R. (2000). *Refactoring, Improving the Design of Existing Code.* Addison- Wesley, first edition.

Burns, A. and Wellings, A. (2009). *Real-Time Systems and Programming Languages.* Addison -Wesley, forth edition.

Chéramy, M. (2014). Simso - simulation of multiprocessor scheduling with overheads.

Cockburn, H. (2001). *Agile Software Development.* Pearson Education Inc., first edition.

Cohn, M. (2004). *User Stories Applied, For Agile Software Development.* Addison-Wesley, first edition.

Design Patterns, Wikipedia (2015). Design patterns. Available at `http://en.wikipedia.org/wiki/Design_Patterns`.

Douglass, B. P. (2004). *Real Time UML.* Addison-Wesley, third edition.

Gamma, Helm, Johnson, Vlissides (1995). *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison -Wesley, forth edition.

Grenning, J. W. (2011). *Test-Driven Development for Embedded C.* The Pragmatic Bookshelf, first edition.

Guiney, K. . (2003). *Use Case, Requirements in Context*. Addison-Wesley, second edition.

Hasle, T. E. (2008). *Systemutvikling, Applikasjoner og databaser*. Cappelen Akademiske Forlag, first edition.

Laplante, P. A. (2004). *Real-Time System Design and Analysis*. John Wiley, third edition.

Lipari, G. (2005). rtsched latex style. Available at `http://retis.sssup.it/~lipari/software/rtsched.php`.

Mall, R. (2007). *Real-Time Systems: Theory and Practice*. Dorling Kindersley, first edition.

Martin, R. C. (2000). Design principles and design patterns. Available at `http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf`.

McConnel (1996). *Rapid Development*. Microsoft Press, first edition.

Microsoft (2015). Microsoft visual studio. Available at `https://www.visualstudio.com/`.

Robbins, S. (2007). Using the process scheduling simulator. Available at `http://vip.cs.utsa.edu/simulators/guides/ps/ps_doc.html#Processes`.

Rüping, A. (2003). *Agile Documentation*. John Wiley, first edition.

Shaw, A. C. (2001). *Real-Time Systems and Software*. John Wiley, first edition.

Shore, W. (2007). *The Art of Agile Development*. O'Reilly Media Inc., first edition.

University of Padova (2012). Scheduler simulator. Available at `http://schedsimula.sourceforge.net/`.

van Heesch, D. (2006). *Doxygen*.

Weller, J. (2006). Cpu scheduler application. Available at `http://jimweller.com/jim-weller/jim/java_proc_sched/`.

Wiegers, K. E. (2003). *Software Requirements*. Microsoft Press, second edition.

Young (1982). *Real-Time Languages: Design and Development*. Ellis Horwood, first edition.

yWorks (2015). yed graph editor: High-quality diagrams made easy.

Appendix A

Appendix

A.1 Minutes of Meeting - client-developer

Meeting Minutes Report:
Requirement Specifications

Date and Time	Wednesday 04 February 2015 at 13:00
Venue	Room D-231 NTNU
Participants	Sverre Hendseth and Inger Johanne Rudolfsen
Secretary	Inger Johanne Rudolfsen

In this meeting, the student and software developer Inger Johanne Rudolfsen asked the supervisor and client Sverre Hendseth questions to specify the requirements of a scheduler simulation tool. The questions and the answers below represent a summary of what was being said during the meeting.

Questions	**Answers and Notes**
Who is the user?	Typically, a student with sufficient knowledge in C++.
What are the inputs for the simulator?	Schedulers and task-sets. Several types of schedulers can be specified to be compared in performance on the same task-sets. The simulation-tool should be able to generate tasks and task-sets for this purpose. When generating, the user should be able to chose the distribution of the task-sets.
What are the outputs from the simulation?	Relevant plots and graphic representations of the results should be provided. If a deadline is missed, this should be very clearly shown in the outputs. Statistical information about the completion of the task-sets within the respective deadlines ought to be calculated, and should be used to compare different scheduler types.
What separates this simulation tool from other existing tools?	This simulation tool must be able to support existing as well as future scheduler standards. An example of a future scheduling standard is Online Execution Time Analysis (OETA). Existing scheduler simulators can be examined in design and implementation, and be used to improve and help design choices made for our simulation tool.
SimSo is an existing scheduler simulator, that seems to work well. How does this affect the work on the new simulation tool?	SimSo should be examined to discover missing features and shortcomings. These faults must then be improved and put in the new simulation tool.
Is GUI a necessity for this simulator?	GUI is not necessary, but if a GUI would serve a purpose, and there is time, it can be added.
Other remarks	Maybe the tool should support multi-core as well as simple-core. A meeting with Nicholas(other student Sverre Hendseth supervises) should be arranged. Nicholas is currently researching new possible scheduling strategies, like OETA.

Next Guidance Meeting: Monday 09 February 2015 at 14:00.

A.2 Second Development Iteration

A.2.1 System Class Diagram after Second Development Iteration

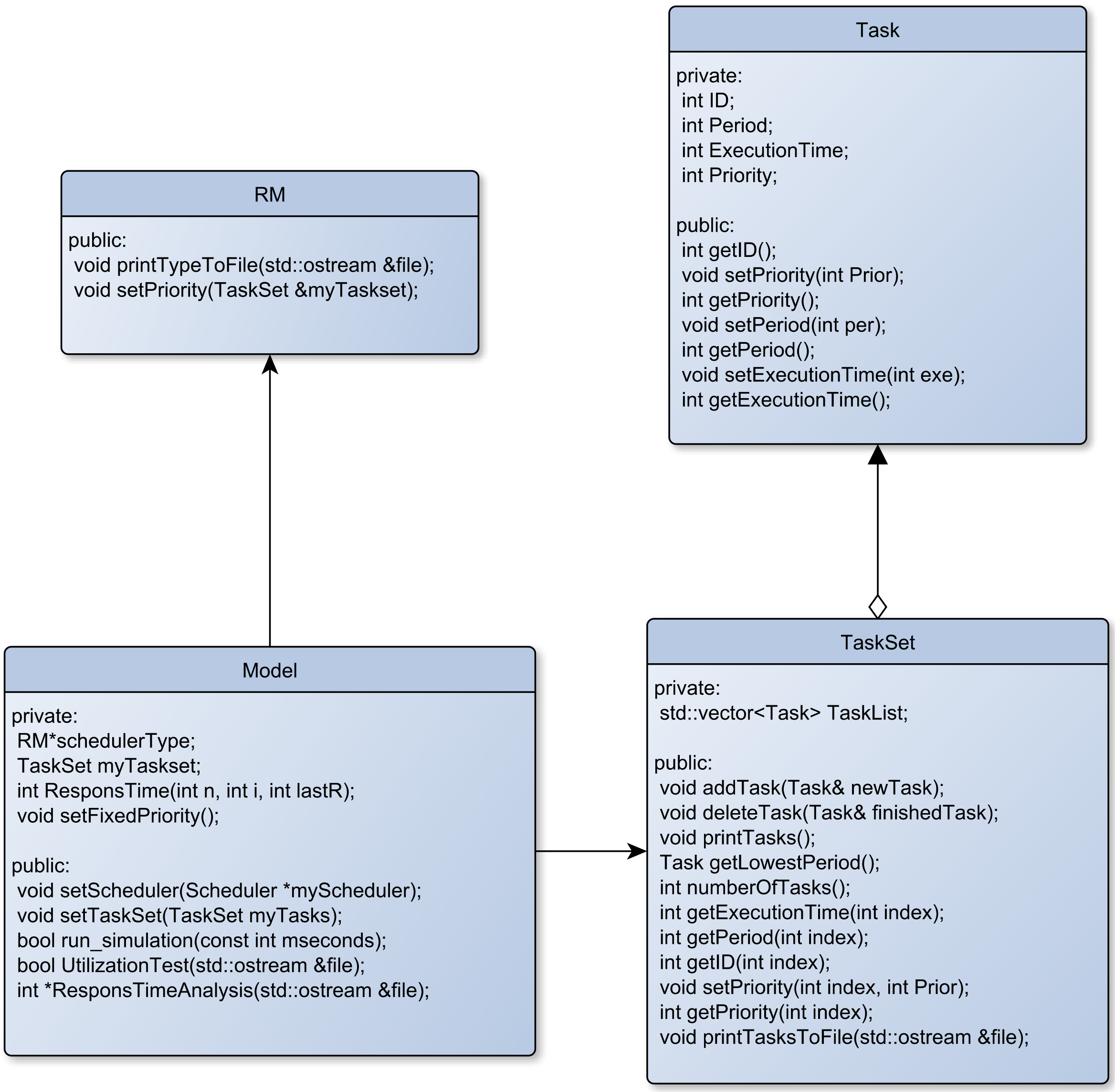
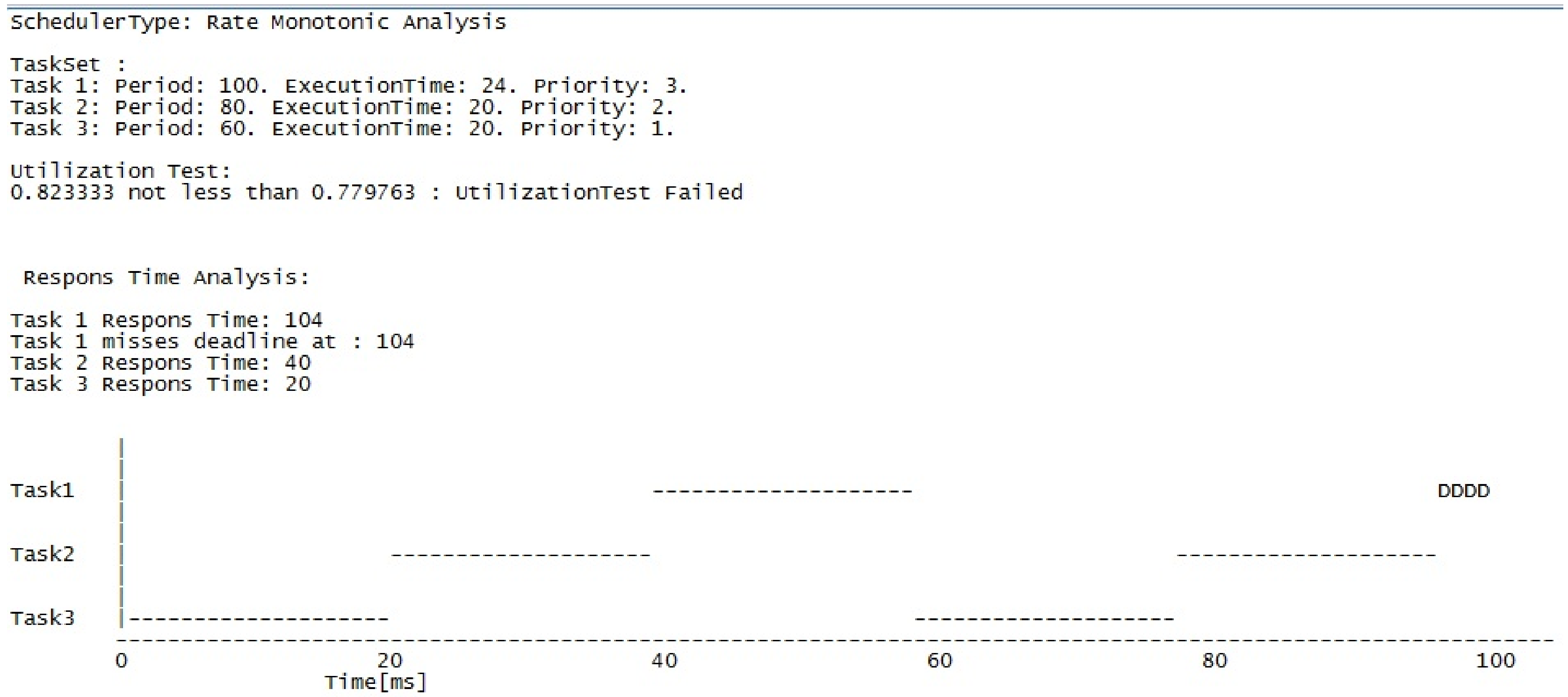

Task
private:
int ID;
int Period;
int ExecutionTime;
int Priority;
public:
int getID();
void setPriority(int Prior);
int getPriority();
void setPeriod(int per);
int getPeriod();
void setExecutionTime(int exe);
int getExecutionTime();
RM
public:
void printTypeToFile(std::ostream &file);
void setPriority(TaskSet &myTaskset);
Model
private:
RM*schedulerType;
TaskSet myTaskset;
int ResponsTime(int n, int i, int lastR);
void setFixedPriority();
public:
void setScheduler(Scheduler *myScheduler);
void setTaskSet(TaskSet myTasks);
bool run_simulation(const int mseconds);
bool UtilizationTest(std::ostream &file);
int *ResponsTimeAnalysis(std::ostream &file);
TaskSet
private:
std::vector<Task> TaskList;
public:
void addTask(Task& newTask);
void deleteTask(Task& finishedTask);
void printTasks();
Task getLowestPeriod();
int numberOfTasks();
int getExecutionTime(int index);
int getPeriod(int index);
int getID(int index);
void setPriority(int index, int Prior);
int getPriority(int index);
void printTasksToFile(std::ostream &file);

Figure A.1: Complete Class Diagram of System after second Iteration

A.2.2 Resulting Output from Second Iteration of Developing the Scheduling Simulator



```
SchedulerType: Rate Monotonic Analysis

TaskSet :
Task 1: Period: 100. ExecutionTime: 24. Priority: 3.
Task 2: Period: 80. ExecutionTime: 20. Priority: 2.
Task 3: Period: 60. ExecutionTime: 20. Priority: 1.

Utilization Test:
0.823333 not less than 0.779763 : UtilizationTest Failed


 Respons Time Analysis:

Task 1 Respons Time: 104
Task 1 misses deadline at : 104
Task 2 Respons Time: 40
Task 3 Respons Time: 20

        |
        |
Task1   |                                         --------------------                                        DDDD
        |
        |
Task2   |                     --------------------                                        --------------------
        |
        |
Task3   |--------------------                                         --------------------
        ----------------------------------------------------------------------------------------------------------------
        0                    20                   40                    60                  80                   100
                         Time[ms]
```

Appendix B

Scheduler Simulator Header Files

B.1 Task.h

```
#pragma once

/****************************************************************
The Task represents a process or thread that is to be executed.

ExecutionTime represents the estimated execution time of the task.

Progression is the time the task has executed since Tstarted.

Tarrival is the time the task is submitted to the system.

FirstTarrival is the original value of Tarrival, created to reset
the task later, as Tarrival is changed and updated for periodic
tasks.

Tstarted is the time the execution of a task is started.

Deadline is the relative deadline of the task.

Period is the period of a periodic task. If the task is non-periodic,
the period is zero.

State is the task state in its life cycle, which can be:
IDLE, READY, RUNNING, WAITING, BLOCKED or FINISHED

ID is the task identification, or task number. This must be a unique
number, not shared with other tasks in the same task set.
****************************************************************/
```

```
class TaskHandler;
class Simulator;
enum taskState{ IDLE, READY, RUNNING, WAITING, BLOCKED, FINISHED };

class Task
{
    friend class TaskHandler;
    friend class Simulator;
private:
    double ExecutionTime;
    double Progression;
    double Tarrival;
    double FirstTarrival;
    double Tstarted;
    double Deadline;
    double Period;

    int State;
    int ID;

    bool DeadlineMissed;

    void updateProgressionTime(double time);

public:
    Task();
    Task(int id, double tarrival, double deadline,
                            double executiontime, double period);
    Task(int id);
    ~Task();

    int Priority;

    int getID();
    int getState();
    double getPeriod();
    double getDeadline();
    double getTarrival();
    double getExecutionTime();
    double getRemainingExecutionTime();
};
```

B.2 Scheduler.h

```
#pragma once
#include"SimulatorCore/Task.h"
#include"Containers/Set.h"
```

```
/************************************************************
Scheduler is the class defining the interface that connects
the implementation of scheduling logic with the simulator.

availableTaskIterator: Iterator that traverses the tasks
available for execution. Corresponding to a "ready queue."

TaskIterator: iterates through all the tasks, ready or not.

isPreemptive(): Returnes the Preemptive variable value.

setTasks(Set<Task*>* tasks) initialize the iterators.

Schedule(time): A virtual function that must be implemented
by sub-classes, aka. new scheduling policies, containing
the scheduling logic.

************************************************************/


class Scheduler
{
protected:
    AbstractIterator<Task*>* availableTaskIterator;
    AbstractIterator<Task*>* TaskIterator;
    bool preemptive;
public:
    Scheduler();
    ~Scheduler();
    bool isPreemptive();
    void setTasks(Set<Task*>* tasks);

    virtual Task* Schedule(double time) = 0;
};
```

B.3 Model.h

```
#pragma once
#include"Task.h"
#include"../Containers/Set.h"
#include"../Scheduler.h"
#include"TaskHandler.h"


/************************************************************************
The Model contains all the information needed to conduct a simulation.
```

```
TaskSet: A set of tasks to be executed.

Scheduler: A policy to schedule tasks for excecution.

RunTime: The simulation run time.

ContextSwitch: The time-loss due to interrupting a current task being
executed and replacing it with another.

ModelName: The name of the model, naming the result report.

modelTaskHandler: A handler for the task set, updating tasks, etc.
*************************************************************************/


class Model
{
    friend class Simulator;
private:
    Set<Task*>* TaskSet;
    Scheduler* scheduler;

    double RunTime;
    double TimeInterval;
    double ContextSwitch;
    std::string ModelName;
    TaskHandler modelTaskHandler;

public:
    Model();
    Model(std::string modelName, Set<Task*>* modelTaskSet,
                         Scheduler* modelScheduler, double modelRunTime);
    Model(std::string modelName, int numberOfPeriodicTasks,
                    int numberOfAperiodicTask, Scheduler* modelScheduler);
    ~Model();

    void setModelName(std::string modelName);
    void setScheduler(Scheduler* myScheduler);
    void setTaskSet(Set<Task*>* myTasks);
    void setContextSwitch(double msTime);
    void setTimeInterval(double time);
    void setRunTime(double time);
};
```

B.4 Simulator.h

```
#pragma once
```

```
#include"../Monitor/Monitor.h"
#include"Event.h"
#include"Model.h"
#include"../Containers/Queue.h"

/**************************************************************
The Simulator conducts the simulation of the behavior of a
processor executing a task set using a scheduler policy.

The Simulator runs a simulation using a model, containing all
the parameters needed to conductct the simulation.

The simulation is event based, where different events evoces
calls on different functions to handle the event.

***************************************************************/


class Simulator
{
private:
    Model* simModel;
    Monitor logMonitor;


    Task* currentTask;
    Event* currentTaskFinishedEvent;
    int currentEvent;

    Queue<Event*> eventQueue;

    //Simulation parameters
    bool preemptive;
    double timeInterrupt;
    double runTime;
    double contextSwitch;

    //Event-handeling functions:
    void onTimeInterrupt(double time);
    void onTaskReady(double time);
    void onTaskFinished(double time);
    void onSimulationFinished(double time);

    void runScheduler(double time);
    void setUpTaskForExecution(double time);

    //After a simulation, the simulator must reset
    void resetSimulator();
public:
    Simulator();
    ~Simulator();
    int runSimulation(Model* myModel);
};
```

B.5 TaskHandler.h

```
#pragma once
#include"Task.h"
#include"../Containers/Set.h"
#include"../Monitor/Monitor.h"

/**************************************************************************
TaskHandler manages the task set for the Model, and generates
task pointer sets for the Model if needed.

availableTaskIterator: Iterator that traverses the tasks
available for execution.

TaskIterator: iterates through all the tasks, ready or not.

generateTask(int N, bool PeriodicTasks) generates a task set of
N tasks based on normal distribution. PeriodicTasks indicates
if the tasks shall be periodic or not.

resetTasks() resets the tasks in the task set after a simulation.

numberOfTasks() return the number of tasks in the task set.

taskAvailable() returns true if there currently exsists a task
ready for execution.

getTaskClosestInTime(double time) returns the task with tarrival
closest in time.

updateTaskStates(Monitor* myMonitor, double time) updates the task states
corresponding to the current simulation time.

checkForDeadlineBreaches(Monitor* myMonitor, double time) checks if any of
the tasks have missed their deadlines.
**************************************************************************/


class TaskHandler
{
private:
    AbstractIterator<Task*>* availableTasksIterator;
    AbstractIterator<Task*>* taskIterator;
    int idCounter;
public:
    TaskHandler();
    ~TaskHandler();

    Set<Task*> generateTasks(int N, bool PeriodicTasks);

    //initializes the iterators.
    void initialize(AbstractIterator<Task*>* taskIt,
                    AbstractIterator<Task*>* availableTaskIt);

    void resetTasks();
    int numberOfTasks();
    bool taskAvailable();
    Task* getTaskClosestInTime(double time);
    void updateTaskStates(Monitor* myMonitor, double time);
    void checkForDeadlineBreaches(Monitor* myMonitor, double time);
};
```

B.6 Event.h

```
#pragma once

/*************************************************************
The Event describes the different events that can occur
during a simulation, and at what time they are set to
happen.
*************************************************************/


enum eventType{ TimeInterrupt, TaskReady, TaskFinished, SimulationFinished };

class Event
{
private:
    int EventType;
    double EventTime;

public:
    Event(int tempEvent, double tempTime);
    Event();
    ~Event();

    int getEventType();
    double getEventTime() const;
    void setEventTime(double tempTime);
    bool operator<(const Event& first)
    {
        return (EventTime < first.getEventTime());
    }
};
```

B.7 Monitor.h

```
#pragma once
#include <fstream>
#include<string>

#include"../SimulatorCore/Task.h"
#include"LogEvent.h"
#include"../Containers/Set.h"
#include"../Containers/Queue.h"

/*************************************************************************
The Monitor keeps track of the events that occur during a simulation.
A report can be generated after the simulation has completed.

"taskIDs" contains all the IDs for the tasks currently being
simulated by the simulator.

"LogEventQueue" contains the logged events(LogEvent objects) from
 the simulation.

"myfile" is the outstream to the tex file that is going to be used
 to generate a pdf-file to present resulting relevant information
 from the simulation.

"name" is the name of the Model currently being simulated, being
 used as a name for the resulting report pdf file.

Functions starting with "log" are used by the simulator to log events.

"generateReport()" generates a report containing relevant information
 about the simulation. The function loops through the LogEventQueue,
 and handles the LogEvents with appropriate functions.

The resulting report contains:
- The Model Name
- A list of tasks and their parameters
- Average TurnAround Time
- CPU idle time
- Number of Deadlines missed and reached
- Number of tasks executed at least one time.
- Gantt chart.
*******************************************************************************/


class Monitor
{
private:
    std::vector<int> taskIDs;
    Queue<LogEvent> LogEventQueue;
    std::ofstream  myfile;
    std::string name;

    //The RT box is used to create Gantt charts
    void createRTBox(int numberOfTasks, double time, int iteration);
    void endRTBox();


    //functions to handle logEvents when generating report:
    void onArrived(LogEvent log);
    void onStarted(LogEvent log);
    void onBlocked(LogEvent log, double time);
    void onFinished(LogEvent log, double time);
    void onDeadline(LogEvent log);
    void onDeadlineMissed(LogEvent log);

    //Calculate TurnAroundTime, CPU IDLE time, DeadlineMisses...
    void calculateSimulatorStatistics(double time);
    //... using these functions:
    double CPUidleTime();
    double numberOfDeadlinesReached();
    double numberOfDeadlinesMissed();
    double AverageTurnAroundTime(int taskid);
    double numberOfTasksCompleted();

    void runLatex();

public:
    Monitor(){};
    void Initialize(std::string resultsname);
    ~Monitor();

    //Functions to log events and information
    bool logTasks(Set<Task*> tasks);
    bool logArrivalTime(Task arrTask, double arrTime);
    bool logStart(Task startTaskId, double time);
    bool logPause(Task pauseTaskId, double time);
    bool logEnd(Task endTaskId, double time);
    bool logDeadline(Task deadlineTask, double time);
    bool logDeadlineBreach(Task deadId, double time);
    bool logSimEnd(double time);

    void generateReport();

};
```

B.8 LogEvent.h

```
#pragma once
#include"../SimulatorCore/Task.h"

/*********************************************************************************
The LogEvent represents an event that has been logged by the
Monitor.

logTask represents the task the logEvent concerns.

logTime is the time of the logEvent.

logEventType is the type of event. The different LogEventTypes are:
    logARRIVED, logSTARTED, logBLOCKED, logFINISHED,
    logDEADLINE, logDEADLINEMISSED and logSIMEND
*********************************************************************************/

enum logEvent{logARRIVED, logSTARTED, logBLOCKED, logFINISHED,
                     logDEADLINE, logDEADLINEMISSED, logSIMEND};


class LogEvent
{
public:
    Task logTask;
    double logTime;
    int logEventType;

    LogEvent();
    LogEvent(Task myTask, double mytime, int type);
    LogEvent(double mytime, int type);
    ~LogEvent();

    bool operator==(const LogEvent& b);
    bool operator<(const LogEvent& first)
    {
        return (logTime < first.logTime);
    }
};
```

B.9 AbstractContainer.h

```
#pragma once
#include"../Iterators/AbstractIterator.h"

/*******************************************************
This is an abstract class specifying the main interface
for interaction with the containers, aka its sub-classes.
*********************************************************/

template<class Item>
class AbstractContainer
{
public:
    virtual AbstractIterator<Item>* createIterator() = 0;
    virtual void addItem(Item a) = 0;
    virtual void remove(Item a) = 0;
    virtual long numberOfItems() = 0;
    virtual Item getItem(long index) = 0;
};
```

B.10 Queue.h

```
#pragma once
#include<list>
#include"AbstractContainer.h"
#include"../Iterators/GeneralIterator.h"
#include"../Iterators/SortedQueueIterator.h"


/***************************************************************
 This is a container, a sub-set of "AbstractContainer" class.

 This Queue Container can be sorted, if the template Item
 can be compared as in the funtion "compareItems".

 The function "emptyQueue()" clears the content of the container.
*****************************************************************/


template<class Item>
class AbstractIterator;
```

```
template<class Item>
class SortedQueueIterator;

template<class Item>
class GeneralIterator;


template<class Item>
bool compareItems(const Item& lhs, const Item& rhs)
{
    return (*lhs<*rhs);
}


template<class Item>
class Queue :
    public AbstractContainer<Item>
{
private:
    std::list<Item> myItems;
public:
    AbstractIterator<Item>* createIterator();
    void addItem(Item a);
    long numberOfItems();
    void remove(Item a);
    Item getItem(long index);
    void sortQueue();
    void emptyQueue();
};
```

B.11 Set.h

```
#pragma once
#include<vector>
#include"AbstractContainer.h"
#include"../Iterators/GeneralIterator.h"
#include"../Iterators/AvailableTasksIterator.h"


/*************************************************************
This is a container, a sub-set of "AbstractContainer" class.

Using a vector as base container, elements are easily accessed.

The iterator AvailableTaskIterator can be created if Task
objects are being stored.
*************************************************************/
```

```
template<class Item>
class AvailableTasksIterator;

template<class Item>
class GeneralIterator;

template<class Item>
class AbstractIterator;



template<class Item>
class Set :public AbstractContainer < Item >
{
private:
    std::vector<Item> myItems;
public:
    void addItem(Item a);
    long numberOfItems();
    void remove(Item a);
    Item getItem(long index);
    AbstractIterator<Item>* createIterator();
    AbstractIterator<Item>* createAvailableTasksIterator();
    void addItems(Set<Item> items);
};
```

B.12 AbstractIterator.h

```
#pragma once


/*********************************************************
This is an abstract class specifying the main interface
for interaction with the iterators, aka its sub-classes.
**********************************************************/

template<class Item>
class AbstractIterator
{
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() = 0;
    virtual Item CurrentItem() = 0;
    virtual double NumberOfItems() = 0;
protected:
    AbstractIterator(){};
    ~AbstractIterator(){};
};
```

B.13 GeneralIterator.h

```
#pragma once
#include"AbstractIterator.h"
#include"../Containers/AbstractContainer.h"

/************************************************************
The GeneralIterator can traverse a container inheriting the
AbstractContainer class. Can be used to traverse all elements
in the container.
*************************************************************/



template<class Item>
class AbstractContainer;

template<class Item>
class GeneralIterator :
    public AbstractIterator<Item>
{
public:
    GeneralIterator(AbstractContainer<Item>* items);
    void First();
    void Next();
    bool IsDone();
    Item CurrentItem();
    double NumberOfItems();
private:
    AbstractContainer<Item>* myItems;
    long current;
};
```

B.14 AvailableTaskIterator.h

```
#pragma once
#include "AbstractIterator.h"
#include"../Containers/AbstractContainer.h"

/***************************************************
The AvailableTaskIterator is an iterator for a Set
```

```
containing Task object pointers.

This iterator will only point to tasks available for
execution, that is: tasks with the states:
READY, RUNNING or WAITING.
*****************************************************/

template<class Item>
class AbstractContainer;

template<class Item>
class AvailableTasksIterator :
    public AbstractIterator < Item >
{
public:
    AvailableTasksIterator(AbstractContainer<Item> *items);
    AvailableTasksIterator(){};

    void First();
    void Next();
    bool IsDone();
    Item CurrentItem();
    double NumberOfItems();

private:
    AbstractContainer<Item>* myItems;
    long current;
};
```