SVERRE HENDSETH

LECTURENOTES, TTK4145 REAL-TIME PROGRAM-MING

Contents

Document Introduction

This document will be updated at irregular intervals during the semester to reflect the real-world execution of the course.

Check last years lecturenotes if you want to see further into the future.

Course Introduction

Welcome, toplevel learning goals

Course Learning Goals, TTK4145, 2016

- General maturation in software engineering/computer programming.
- Ability to use (correctly) and evaluate mechanisms for shared variable synchronization.
- Understanding how a deterministic scheduler lays the foundation for making real-time systems.
- Insight into principles, patterns and techniques for error handling, consistency and fault tolerance in multi-thread / distributed systems.
- Knowledge of the theoretical foundation of concurrency, and ability to see how this can influence design and implementation of real-time systems.

1

Multicore important at the time: We **should** know how to do this! Will learn important basics here also.

Large parts of the industry does not know this - And need arise, as demands rise and multicore and wireless expands.

More detailed learning goals: Exist also for all parts of the course: Use them when preparing for the exam?

Question: What is the purpose of the course?

Quality system

Quality System

- Check https://innsida.ntnu.no/wiki/-/wiki/Norsk/Kvalitetssikring+av+utdanning
- I hope you give me constructive and immediate feeldback! Other feedback goes to the quality system in the previous point.
- We are having a questionaire to be answered by everybody at the end of the semester.
- A colleague will be watching one of the lectures to share experiences.
- We are having a reference group (3+ meetings, pizza at the last one) (See next slide)
- "Course reports" will be made, and the department follows up on improvements. Old reports are available for anybody.

1

Reference Group

Reference Group

...Someone to represent you, and for me to consult...

- One from Kyb 5year
- One from Kyb 2year
- One not from kyb (elsys, idi, marin, maskin,... ?)
- One international
- One woman
- More categories? Do you want to contribute? Who do you want to represent you?

Three meetings (one of you writes the minutes...) Pizza on exam day :-)

1

Activities & Grading

Lectures, exercises, project, exam.

Mandatory activities & Grading

- Exercises are mandatory, to be approved on the lab by the student assistants.
- A project (control 3 cooperating elevators) evaluated in 3 parts (Design, code quality, funcionality). Counts for 25% of your grade.
- An exam at the end of the semester counting 75%.

1

Lab Groups

Lab Groups

Groups of 3; We decide the groups.

- Pedagogical gain from student cooperation is clear (or at least commonly accepted at the time by both Ntnu and students).
- "Random groups" are fair, more challenging and you have more to learn from people you do not know well already.
- A "low ambition" is harder to negociate in a random group. (Similarily; "I am best at this so I should make decitions" is not so viable).
- While work possibly will be more tense, it will not take more time. Maybe even the explicit ambition discussion will reduce work?
- How can we further stimulate you learning from each other?

1

Semesterplan

List of topics per week

Topics per week

Week	Topic
2	Course intro and Project startup
3	Project; Languages, Design, Code Quality, Network, Fault Toler
4	Code Quality, Ada
5	Fault Tolerance Basics
6	Transactions
7	Atomic Actions
8	Asynchronous transfer of control, Exceptions?
9	Shared Variable Synchronization (Semaphores)
10	History and Introduction
11	Monitors, Ada and Java
12	Scheduling
13	Formal methods and messagebased systems
14	Guest Lecture

1

Learning resources

Learning Resources

... there are more than you can appreciate ...

- Lecture videos from 2016
- This years lectures
- Lecture notes 2017, lecture notes 2019
- recommended reading (pdfs) on blackboard
- Old exams (Skip 2018?) (I do not intend to make major changes)
- Learning Goals
- https://www.wikipendium.no/TTK4145_Real_Time_Programming
- http://www-ist.massey.ac.nz/csnotes/355/lectures/monitors.pdf
- ...
- Inform me if you find something recommendable!
- The internet

1

Three safe ways

3 safe ways?

— Learning-goal based: Use the learning goals and old exams to get a feel for what you need to master.
— Video-based. The videos from 2016 is a complete (until further notice) walkthrough of the learning goals.
— Lecture-based. The lectures of 2019 will not be a complete walkthrough, but will create awareness enough for you to see what you need to master. Then ask!

See the ttk4145_sporreundersokelse.pdf document on blackboard.
But be aware: Some of the learning goals are quite deep, going even beyond knowledge and skills. This requires ... maturation.

1

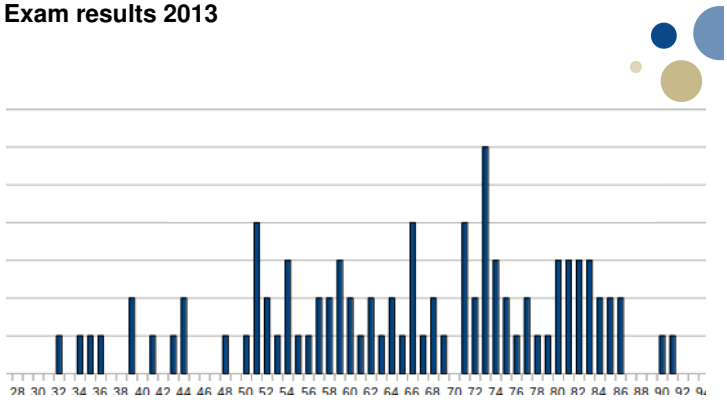
Task: Recognize numbers in stream of characters.

Task: Recognize numbers in stream of characters

— You read a text file character by character (char getNextChar()) and are to pick out and print all the numbers.
— Legal numbers: 1,2,3,-234,1.2,.05,-0.43,1e-12,-199.23E98,.4e+98,...
— How would you structure this small program?

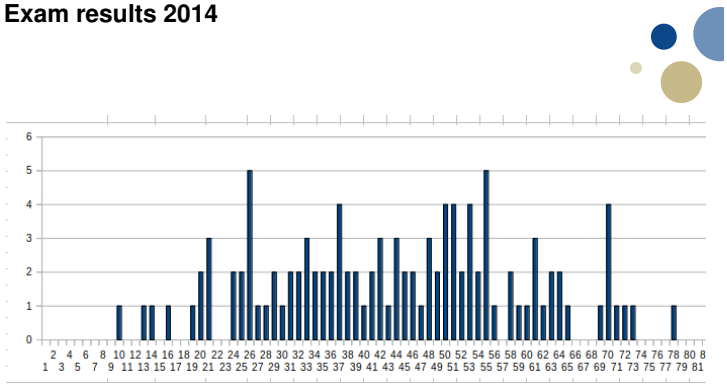
1

Some Grade statistics 2013



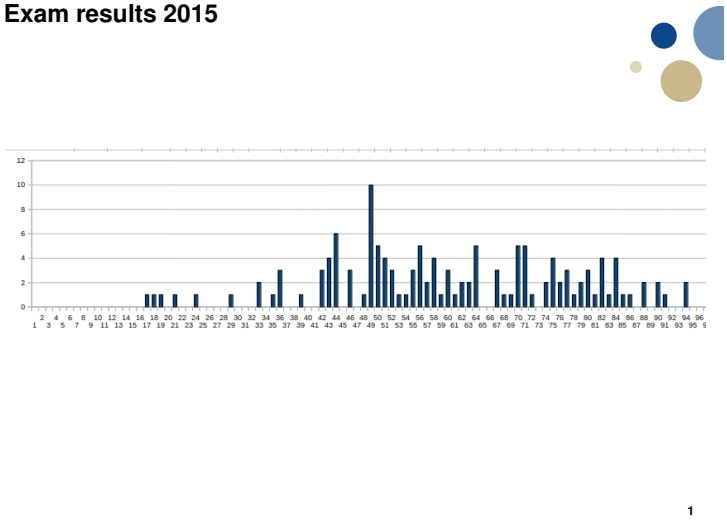
1

Some Grade statistics 2014

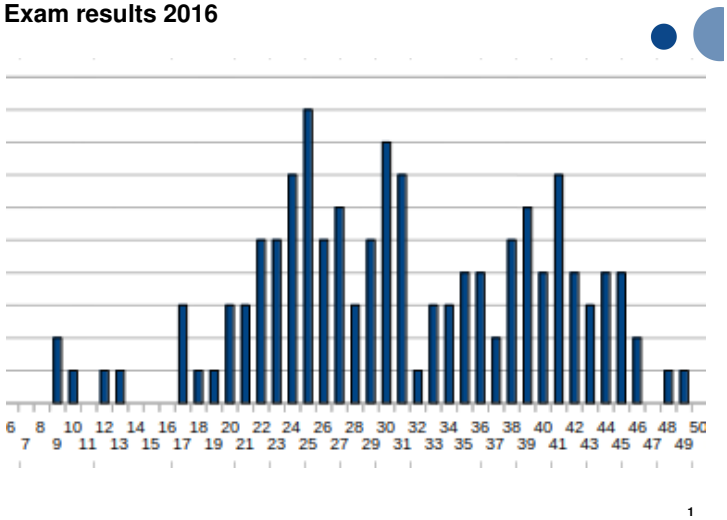


1

Some Grade statistics 2015

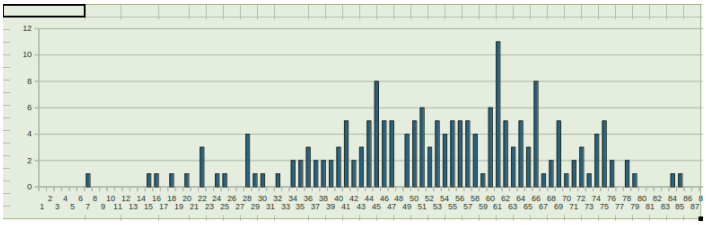


Some Grade statistics 2016



Some Grade statistics 2017

Exam results 2017



1

Study skills: How to learn Abstractions and Thought patterns

Study skills: How to learn Abstractions and Thought patterns, including SW Design

- Learn-by-example. Be with role models.
- When the number of examples reaches a treshold, you will abstract.
- Creating awareness, Setting focus
- Passively notice examples.
- Actively notice examples.
- Interview those that can solve problems you can not.
- Use teachers, Explain what you do not understand, Ask questions.

Is there a market for a walkthrough of these?

1

Filming

Recordning this years lectures

— Anybody against filming?
— (Some lectures *will* be filmed - those potentially useful for me next year.)

But lectures this spring will not have any ambition of being a complete walkthrough...

— ...and filming is not longer supported by Ntnu... Meaning quality/success rate will be limited.
— Would you still appreciate availability of such (sparse?) "for the record" videos?

1

Fault tolerance teaser.

Fault tolerance teaser

If you have a bug in your SW or if a "cosmic ray" at any time flips any bit in your computers memory; Is it still possible to make SW that behaves according to spec?

1

Blackboard Calls

Week 2 Call: Course Intro / Project

Hi all,

The lectures in Real-Time Programming starts Monday (12:15-15:00 in EL3). This week we will likely also use the lecture/exercise timeTuesday (14:15-16:00 in KJL1).

We will spend the first 2-3 weeks, and this week especially, giving the project work a good start.

The project starts immediately, and you are expected to work on the project in parallel with the exercises.

Tomorrow, in addition to this and other practical details of the course, we may give an overview/summary of the fault tolerance part ofthe course and start getting into code quality considerations.

I aim, now, for no major changes in curriculum (or exam format) compared to last time I taught this course (2017). Still, the course is evolving:

This year we will put together lab groups differently from before: The default group size will be three, and the groups will be decided by us. Since forming groups is slightly more complicated, the mandatory exercise 1 is to be done individually. After my lectures were filmed in 2016, these films represent a fair walkthrough of the curriculum, leaving freedom to spend lecture time in other ways. How this is best done is subject to discussion and change. Check the "How to feel safe" chapter in ttk4145$_{\text{sporreundersokelse.pdf}}$ (under "Course information")! Use this opportunity to give the semester a great start!

See you!

Sverre

PS: There may be a kahoot; bring a device.

PS2: We are working on the course information on blackboard still. Be forgiving.

Week2 Call - part 2 (Tuesday): Intro to multithreading

Tomorrow, I will actually do the live demo, then talk more about the why and how we write multithreaded programs. We will discuss the difference between concurrency and parallelism, touch on the basics of message passing and synchronization, and maybe have some tangents on hardware architecture and code quality.

By the way, the old "natural language" project specification is found here: `https://github.com/TTK4145/Project-2017` Take a look at it and compare. i'm confident that there is a "correct" solution for how to present this kind of information. If we have time at the end, we'll have a go at "logos in action" to see if we can find what the answer really is. I want you to contribute here, so make some mental or actual notes if you have the opportunity.

- Anders

Week 3 Call: Sverres intro to project and course summary

Hi all,

Monday I will give my introduction to the project, and, in the project-relevant perspective, give a summary of the whole course:

The fault tolerance techniques presented in the course will be listed and shortly explained. (These techniques represent represents "best practices" when solving problems like the project.) Also some common, good, and very different from each other, design solutions to the project will be mentioned.

Building on Anders' introduction to multithreading and semaphores tuesday I will introduce extensions and pitfalls with shared-variable synchronization.

Finally, the alternative way of thinking on multithreaded systems; messagepassing will be motivated and introduced. This segment of detailed-design solutions is what motivates the use of Google Go (Golang) and the other messagepassing languages in the project.

See you!

Sverre

Week 4 Call: Software quality

Hi all,

The theme for tomorrow will be code quality. Not only do we go through what is expected of you in the project (1/3 of project evaluation is on code quality), and how to answer code quality related exam questions. Code quality is also the criteria for evaluating tools, language mechanisms, techniques, designs etc. in all parts of later lectures: Why are global variables bad? What are the downsides to using semaphores? Which is the best programming language for shared variable synchronization? When is shared variables better than message passing? - It all boils down to what yields the higher code quality.

Some chapters (6,7,11,32) from a fabulous book, Code Complete, is recommended reading for this part. Buy the book - it fits in any programmers bookshelf.

The 2016 videos represents a walk-through of the curriculum. On Mondays lectures I will try to convey the connection between the underlying abstraction "Code Quality" and any concrete advice or rules to enable you to ... make up your own opinions rather than rely on advice, rules-of-thumb and checklists.

If you want to prepare for tomorrow; Each Code Complete chapter has some checklists and "key points" summary (google "code complete checklists"); look through them.

I may not use all three hours; Prepare your questions - about anything.

See you,

Sverre

Week 5 Call: Basic Fault Tolerance

Hi all,

Tomorrow we will start at the fault tolerance part of the course. I have given a two-hour overview of the techniques most immediately relevant for the project. Now I will try to connect these to reality by discussing them in the context of a possible elevator design.

Topics that will be discussed, (in addition to the project and the design process) will be

- Fault Tolerance (The term)
- Error modes
- Static and Dynamic redundancy
- Error detection
- Forward and Backward Error Recovery and the domino effect.
- Acceptance Tests
- Merging of error modes
- Writing checkpoints, checkpoint-restart

I may not use all three hours; Prepare your questions - about anything.

See you,

Sverre

Week 6 Call: SW fault masking

Hi all,

I think that even more training on the "fault tolerance way of thinking" is appropriate.

Tomorrow I will therefore go through the three examples in the Gray chapter on "Fault Model and Software Fault Masking" (including process pairs), and continue my design of an elevator system from where I left off last time.

I will also go through the criteria for the upcomming design review.

I may not use all three hours; Prepare your questions - about anything.

See you,

Sverre

Week 7 Call: Ada

Hi all,

Monday 26/1 Kristoffer Gregertsen from SINTEF will give an introduction to the Ada programming language.

Ada is a large and carefully designed language, well suited for embedded software and some of the choices made on how to do timing and thread interaction are very interesting (and relevant for the exam). These mechanisms will be discussed explicitly in later lectures, but building familiarity with the language, enough for you to understand the Ada examples in the recommended reading chapters, and to write small Ada programs (or "Ada pseudo code" on the exam) happens tomorrow. If you have not settled on the project programming language yet; Maybe Ada?

One or two of the exercises will be using Ada.

See you,

Sverre

Week 8 Call: Atomic Actions and Transactions

Hi all,

We discussed (forward and backward) error recovery two weeks ago, and saw that in our context - with more interacting/cooperating participants - finding the way to recover from an unknown error was not trivial (the domino effect etc.).

For the project you have been able to assume that only one node is restarted at a time, and you have made sure, in your own ad-hoc ways, that a restarted node recovers into a consistent state.

Tomorrow we are loosening these assumptions leaving the project behind: There will be more ongoing operations concerning different subsets of the participants, and recovering from errors will require active cooperation from the participants.

Encapsulating *operations* so that error recovery can be done per-operation rather then per-participant is the trick that leads us to Atomic Actions/Transactions. But these are quite abstract/complex SW design patterns that will be demanding to 'fit into the toolchest'.

The videos from 2016 gives two perspectives on this matter; "Transaction fundamentals" and "Atomic Actions". Tomorrow I will attempt to give a third.

If you have some time to prepare: Familiarize yourself with the learning resources.

The Burns&Wellings "Atomic Action" chapter, the slides under `https://www.cs.york.ac.uk/rts/books/RTSBookFourthEdition.html`

The "Transaction fundamentals" chapter.

The videos from 2016

My 2016/17 lecturenotes and slides on the matter.

I may not use all three hours; Prepare your questions - about anything.

See you,

Sverre

Sverres intro to project and course summary

Introduction: Fault tolerance teaser.

Fault tolerance teaser

If you have a bug in your SW or if a "cosmic ray" at any time flips any bit in your computers memory; Is it still possible to make SW that behaves according to spec?

1

Yes, it is possible. This is what the project is about. (Sverre shall not starve)

It will be a question of probabilities, but the chance of failing can be brought arbitrarily low, and it is relatively straightforward to start calculating what the chances are.

For the project: Just assume that only one error happens at the time: Only one pc looses power, Only one network cable is unplugged etc. A reasonable recovery time of some seconds is allowed. Note: One lost UDP message in a critical instant does not count as an error here - that is to be expected.

Study skills: How to learn Abstractions and Thought patterns, including SW Design

- Learn-by-example. Be with role models.
- When the number of examples reaches a treshold, you will abstract.
- Creating awareness, Setting focus
- Passively notice examples.
- Actively notice examples.
- Interview those that can solve problems you can not.
- Use teachers, Explain what you do not understand, Ask questions.

Is there a market for a walkthrough of these?

1

Remember this; We aim for a level of proficiency here where you can *use* these techniques even in new (sw design) problems - to make them part of your toolchest. (Ref. the test on recognizing numbers last Monday where only a very few of you made the connection to the state machine.)

The embedded systems setting

On the desktop:

- Most user processes are competing — independent of each other.
- Some sense of fairness, responsiveness or performance decides who runs when.
- If one fails it is a local problem — the rest should just continue.

In the embedded system:

- All processes are cooperating — interacting.
- Who runs when should be decided strictly - the most important, or under some regime for satisfying deadlines.
- If one fails it is a system failure.
- *Recovery* from an error often requires cooperation/interaction!

1

Redundancy

Redundancy

Static Redundancy

- Ex. Storing multiple copies of data
- Ex. Resending UDP messages whether they are necessary or not.
- Ex. Having more HW units working in paralell
- Does not work well with SW — If one copy makes an error, the other will also.

Dynamic Redundancy

- If at first you don't succeed, try, try, try again
- ...possibly in another way
- Failures must be detected
- Ex. Resending a message if we time out waiting for acknowledgement.
- Ex. Restart/Reboot and try the same again (Works embarrassingly often)

1

These two should be relatively easy. Detection will be the challenge! How can we detect errors that we did not know might happen?

Detection: Acceptance test

Detection: Acceptance Tests

Do not test for the presence of errors.

Do make tests to check that everything is ok.

A small, but cool, observation:

- **Q:** How can we test that the systems works in presence of unknown errors?
- **A:** Inject failed acceptance tests.

1

This is a hard one: I need something to change in your heads! The reading of the configuration file example:

- All that can go wrong is endless - testing is futile.

Error modes

Error Modes

"Error Modes of X" == "The number of ways the X can fail".

- Error modes is a part of X' interface!
- When designing X, making the error modes few/simple is an important task!
- "I failed" is a good one. "Message was not received".

1

Errormodes of communication frameworks

TCP vs. UDP vs. Broadcast vs. Message queues vs. odd communication libraries and transactional frameworks vs. Erlang vs...

Sverres view on this: It amounts approximately to the same complexity. Choose what suits you. Your challenges must be solved anyway:

- A process may be restarted at (theoretically) any time
- One node may loose network
- The system should never, even theoretically, loose orders.

Find out how to solve these; Then choose communication framework. ... may be a good advice.

1

Merging of error modes

— When you anyway have to handle the worstcase, why not handle all errors in that manner?
— If you detect an error: just restart the program - in spite of the fact that this error might have a less-drastic recovery?
— Were you really interested in *why* reading the configuration file did not result in a consistent configuration?

1

Checkpoint Restart

How to restart?

Some program state is probably necessary to keep over a restart.

1. **Checkpoint restart**. Save the state "occationally". Read the newest old state at startup.
2. **Process pairs**. The new instance of the process has already started, and just receives the checkpoints, ready to take over.
3. **(Running under a transaction-aware OS)**

When is occationally?

— Do all the work, up to and including the acceptance test (but do not do the 'side effects', like giving the reply, lighting the elevator lamp, etc).
— Store a checkpoint.
— Do the side effects.

A newly restarted process will begin by doing (repeating?) the side effects

1

Btw. Storing on disk is uncool

- A power outage may both restart the program *and* trash the disk.
- It is very slow.
- And you need some versioning/checksum, in case a write fails?
- Better let another node store the checkpoint.

The log

The Log

The monolothic checkpoint can be replaced with a sequence of state updates ("Log records").

- The checkpoint can be reconstructed by executing all log records
- If "before-information" is also stored in the log, the ability to run the log backwards enables an elegant UNDO feature that may be useful in error handling.

1

Atomic stuff and transactions

Atomic stuff and Transactions

Atomic stuff is wonderful! Programs will be kept simple (maintainable) since an Atomic Action has no, observable, intermediary state.

- Semaphore operations
- Synchonous communication
- Assembly instructions

We can design such Atomic Actions, typically by locking resouces (so that nobody see the internal state) and syncing the acceptancetest/unlocking between participating threads.
The error modes of such an Atomic Action is a challenge; If we simplify the error modes into "ABORT", then we have a **Transaction**.

1

Threads and processes

Brainstorm on the blackboard: Why do we use threads?

1. "Reflect reality"?
2. One thread per realtime demand
3. Performance

4. System is distributed anyway

5. As a way of dividing the system into modules.

1 is questionable? 2 is central for real-time programming. 3 not us we value predictability over performance. 4 Very relevant (You will have N PCs running programs).

5: Should be the important one for those choosing a messagepassing language.

Threads and Processes

All threads/processes in embedded systems are cooperating. Interaction is typically categorized into

- Synchronization
- Communication

What is best?

- From a SW design perspective: Communication
- From a Real-time programming design perspective: Synchronization is the only option
- (...but be aware. The elevator is not demanding on the RT side. Synchronization may still be viable.)

1

Synchronization vs. Communication

Synchronization vs. Communication

If communication is *conceptually* what you need, then implementing it with synchronization will be more complicated.

And communication most often *is* conceptually what you need...

1

Elevator project, Choosing threads

Elevator project; Choosing threads

Communication based:

— One thread per module. All have while-select structure (see next slide)

Synchronization based:

— NOfThreads == Number of concurrent blocking operations + 1 (=3?)

— But if you teach yourself nonblocking versions of select++ you can manage "without threads".

1

Communication-oriented design

— Think in terms of responsibilities.

— ... and servers yielding services to each other.

All processes have the same structure:

```
func Elevator(ElevStateCh chan  ElevState,...) {
  ...
  for {
    select {
      case temp := <- ElevStateCh:
        ...
      case <-time.After(2*time.Millisecond):
        ...
      ...
    }
  }
}
```

1

Some other project advice and pitfalls

Advice and pitfalls

— Think: Detect what needs handling, then handle it. Preserve the causal relationship.
— (Do not decouple effect from cause by global state. A "what needs to be done" analysis should be concerned only with real things; messages, HW, timeouts, not with state updates.)
— (If nothing needs doing: Do nothing!)
— Do not nest while loops (Also breaks causality).

The underlying rule is; When the system breaks, tracking down the bug in the code should be possible (And the causal relationships are wonderful for this.)

1

Languages in TTK4145

Languages in TTK4145

Three supported languages on the lab:

— **Python:** Python is seen to be lowest-treshold. (not so popular though)
— **C (/C++):** /The/ embedded systems language. Programming C is always useful experience. (But not simplest, and not forgiving)
— **Go:** Go is a supported language because it supports the messagepassing regime. (popular)
— (Sverre: ...but if you make a synchronization/"global state" based design anyway and want to avoid C: You might as well use python.)

Other popular languages: Erlang (also well supported). Let us see more Rust? Ada? (Elixir, Haskell?,...)

Two more languages relevant in lectures:

— **Ada** - have interesting mechanisms for

1

Code Quality

Code Quality Context

Learning goals; Code Quality

- Be able to write software following selected Code Complete Checklists for modules, functions, variables and comments
- Be able to critizise program code based on the same checklists

1

Notice; The 2019 explicit aim is to go beyond the checklists! Check the call:

The 2016 videos represents a walk-through of the curriculum. On Mondays lectures I will try to convey the connection between the underlying abstraction "Code Quality" and any concrete advice or rules to enable you to ... make up your own opinions rather than rely on advice, rules-of-thumb and checklists.

You must *ask the question*! Is this code good? This builds the awareness that lets you track your experiences down to advice.

Ex: "How to name channels" is not covered by the checklists, but relevant for the exam.

Brainstorm: What is Code Quality?

What is the difference between high and low quality code? Any metrics, techniques, principles, signs, rules, monitors, etc.?

1

Comments add to the project size. Must be maintained.

- Modules

- Abstractions

Evaluating the project

Use ~10 min to find the entry point of the code and answer the following questions:		
1	Does the entry point document what components/modules the system consists of? • You can see what threads/classes are initialized	(0-1)
2	Is the connection between modules clear from looking at the entry point? • You can see how different modules interact • You can (with relative ease) find out how modules depend on each other • Circular dependencies are avoided • If there are any global variables, their use is clear and are their names are excellent	(0-3)
Use ~15 min to answer the following questions about modules:		

3	Do the modules minimize accessibility to their internals?
	• Accessibility is enforced programmatically
	  • That is, programming features (or conventions) for "private" and "public" are used
	  • (Go: Lower case fn names, Erlang: `-export([])`, C: fn declarations in .h-files, etc.)
		(0-1)

1

One Code Quality Projection 1: "If you can understand"

Code Quality Projection 1: Can you read it?

If I can look at a page of your code and understand what it does, it is good! (C)

If I can look at a page of your code and see that it is correct, it is perfect! (A+)

1

Minimize the number of assumptions that must be made to say it works. (Global variables, goto, semaphores are bad! Assumptions are mostly assumptions about the rest of the code: Minimizing interfaces and interactions are good!)

Minimize the time it takes to be able to read it. (But note: We *are* willing to invest some effort - optimal function size is ~172)

Example: Casette Player

```
bool play;
bool record;
bool pause;
bool ff;
bool rw;

play(){
  if(play == false && kasett()){
    if(ff || rw){
      mekanikk_stop();
      ff = false;
      rw = false;
    }
    mekanikk_play();
    motor_play();
    if(record == false){
      elektronikk_play();
    }
    play = true;
  }
}
```

1

Example: Casette Player

```
static enum {State_Stop,State_Play,...} g_state;

play(){
  switch (g_state) {
    case State_Open: break;
    case State_Stop: action_play(); g_state = State_Play; break;
    case State_Pause: action_play(); break;
    case State_Rewind:
           action_stop(); action_play(); g_state = State_Play; break;
    case State_FastForward:
           action_stop(); action_play(); g_state = State_Play; break;
    case State_Play: break;
    case State_RecordReady: action_record(); g_state = State_Record; break;
    case State_Record: break;
    case State_RecordPause: action_record(); g_state = State_Record; break;
  }
}
```

1

What are the assumptions here?

Example: Bounded buffer with semaphores

```
SEMAPHORE NInBuffer(0),NFree(N),Mutex(1);

put(e){                        get(e){
  wait(NFree);                   wait(NInBuffer);
  wait(Mutex);                   wait(Mutex);
    // enter into buffer           // get e from buffer
  signal(Mutex);                 signal(Mutex);
  signal(NInBuffer);             signal(NFree);
}                              }
```

1

Is it good? Does it work (How many minutes does it take to convince yourself?) What is the problem?

Example: A Hard to find bug

What do these two functions do? Do they work?

```
void allocate(int priority){
  Wait(M);
  if(busy){
    Signal(M);
    Wait(PS[priority]);
  }
  busy=true;
  Signal(M);
 }
```

```
void deallocate(){
  Wait(M);
  busy=false;
  waiting=GetValue(PS[1]);
  if(waiting>0) Signal(PS[1]);
  else{
    waiting=GetValue(PS[0]);
    if(waiting>0) Signal(PS[0]);
    else{
      Signal(M);
    }
  }
}
```

1

Take the time to find the bug?

```
#include "config.h"
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

#include <math.h>
#include <pthread.h>
#include <string.h>
#include <time.h>
#include <unistd.h>

#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <libheis/elev.h>

#include "comms.h"
#include "messages.h"
#include "operator.h"
#include "orderlist.h"
#include "target.h"
```

```
#define N_ORDERS 3

bool orders[N_ORDERS][N_FLOORS];
void command_signal_callback(int floor);
int fdout, fdin;


int main(){
  run_system();
}
```

1

Comments to explain complex code

If your code is hard to understand:

1. Fix it: Give functions and variables better names so we can see what happens.
2. If it is still hard: The code is bad; Throw it out and rewrite.
3. If that was not viable: There is something wrong with the context the code exists in: Fix the datastructures/design/architecture. (It is worth it!!!)
4. Well, *maybe* the problem you were solving was of a kind that had no readable solutions. Accept this very reluctanly and sorrowfully (like "Hell is real") because it means that if the problem was slightly harder it would be beyond you to solve.

1

Did you notice that there were no mention of comments here?

One Code Quality Projection 2: "Maintainability"

Code Quality Projection 2: Maintainability

If a system is maintainable you can:

— easily find and fix bugs without adding new ones.
— easily add features without compromising existing design/structures.

A metric: Expected effort to fix the next bug.

1

Story: bugs per line (1/50, 1/500, and then 1/550). Story: 1/5 of total development effort so far goes into maintenance per year. (What must this factor be to manage the project on time? Somebody do the sensitivity analysis?)

For your life? Devoted to maintenance (fixing your own bugs) or jumping from development project to develoment project

This matters!

A meaningful module

A meaningful module:

— You should be able to maintain the module without knowing anything about its users or usage patterns.

— You should be able to use the module without knowing details of its internals.

— Ideally: The principle you used for dividing the system into modules should allow **composition**: That supermodules can be made from modules.

Strong cohesion: The module has a purpose or responsibility that its parts contribute to. **Weak Coupling:** The dependencies between modules are simple, easy to understand.

1

Modules may be anything:

- Some lines of code with a blank line over and under, and with a comment as headline.
- A function
- A collection of functions
- A library (collection of collections of functions)
- An Object
- A thread!
- A process
- A Transaction
- A filter, reading from input and emiting to output
- Etc.

Module Interfaces

Module Interfaces

- shall be as 'thin' as possible; Avoid duplicating functionality and convenience functions
- shall not contain assumptions on usage patterns
- must be complete
- (The hidden complexity of the implementation shall not be insignificant)

1

What is the problem here?

```
void InitializeCommandStack();
void PushCommand( Command command );
Command PopCommand();
void ShutdownCommandStack();
void InitializeReportFormatting();
void FormatReport( Report report );
void PrintReport( Report report );
void InitializeGlobalData();
void ShutdownGlobalData();
```

1

Epilogue: Some questions for the class

Epilogue: Some questions for the class

- Why should limiting scope of variables to a minimum be good?
- Why is 'goto' bad?
- ...and btw. what do we think about exceptions?
- Why are preserving causal relationships good?
- Why aim for a minimum state (not have variables for floor, nextfloor, prevfloor, moving and direction)?
- Why is building/choosing a good abstraction important?
- Is C++ a good programming language?

1

Basic Fault Tolerance

Learning Goals

Learning goals; Fault Tolerance Basics

- Understand and use terms (like): Reliability. Failure vs fault vs error. Failure modes. Acceptance test. Fault prevention vs. tolerance. Redundancy, Static vs. Dynamic. Forward/ Backward error recovery.
- Understand, use and evaluate techniques (like): N-version programming. Recovery blocks. Error detection. Failure mode merging. Acceptance tests.

1

Fault Tolerance Motivation

Testing is not good enough

Failure modes of fopen() call from clib:

On error NULL is returned, and the global variable errno is set:
EINVAL The mode provided to fopen was invalid. The fopen function may also fail and set errno for any of the errors specified for the routine malloc(3). (ENOMEM) The fopen function may also fail and set errno for any of the errors specified for the routine open(2). (EEXIST, EISDIR, EACCES, ENAMETOOLONG, ENOENT, ENOTDIR, ENXIO, ENODEV, EROFS, ETXTBSY, EFAULT, ELOOP, ENOSPC, ENOMEM, EMFILE, ENFILE)
Welcome to the dark side

1

Traditional Error Handling

```
FILE *
openConfigFile(){
  FILE * f = fopen("/home/sverre/.config.cfg","r");
  if(f == NULL){
    switch(errno){
      case ENOMEM: {
        ...
        break;
      }
      case ENOTDIR:
      case EEXIST: {
        // ERROR!
        break;
      }
      case EACCESS:
      case EISDIR: {
        ...
        break;
      }
      ....

    }
  }
  return f;
`
```

1

%Bugs: % Do all systems have them? % Yes... % "1bug/50 lines before testing" % "1/500 at release" % "1/550 after a year, and then constant"

The normal way of handling bugs:

- Make your program to fill the (functional) specification
- Run/Test the program
- Errors happen
- Find "cause" in code
  - erroneous code

  - missing handling of some situation
  - incomplete spec

  % My experiences: of 117 bugs 2 were 1 100 3. % The book % Årsaker til bugs: % Feil/Uklarheter/mangler i spec. % SW bugs % HW %Situasjoner som burde vært håndtert. % Nettverk/kommunikasjon

- Add/Change Code: Fix code or Detect/Handle situation % "rette feil" skjer overaskende skjelden.

  This is not good enough!

- We must handle unexpected errors

- Have more participating threads in error handling.

- Testing is not good enough
  - Can only show existence of errors
  - Cannot find errors in spec.
  - Is realistic testing possible?
  - Cannot find race conditions
  - World interface: - can realistic testing be done (must simulate world...)
  - Embedded systems may have higher demands of safety

How can we handle unexpected errors, transient errors, and yet unfixed errors? % How can we handle the bugs that are still left in the system after testing? % (Ref. Cosmic Rays?)
Answer: Fault Tolerance! % Chapter headline!)
Fault-Error-Failure -> definition of Fault Tolerance

What is a bug Most likely an unhandled situation!
Let us discuss SW Specs

Terms and techniques

Terms and techniques

- Fault Tolerance (The term)
- Error modes
- Static and Dynamic redundancy
- Error detection
- Forward and Backward Error Recovery and the domino effect.
- Acceptance Tests
- Merging of error modes
- Writing checkpoints, checkpoint-restart

1

The Design Process

Many thick books has been written about systematic approaches to software design...

Here is my take:

1. Have a brainstorm on "what the system must handle": You must handle "that a button is pressed", "that a node is killed", etc. This need not be a complete set of scenarios, but should be illustrating different aspects of the system functionality. (I would guess 4-8 scenarios should span the functionality space)

2. Divide the system into modules based on your gut feeling and discussions with your partner.

3. Map the scenarios onto the modules in a "then this must happen" manner: "The button press is detected by the X module, that notifies the event loop in the Y module. Then a message is sent ... etc." (Also here: do not give in to the temptation to make "complete" scenarios, focus on spanning the space.)

4. Draw the diagram over module interactions. Who calls who?

5. Sum up, for each module, all its incomming interactions/requests. These are the specification *of the module*.

6. Try to make a perfect module interface that fulfils the modules responsibilities.

7. Move responsibilities between modules (reorganize how the system is divided into modules if necessary) so that the diagram in 4. gets fewer arrows and that the module interfaces in 6. becomes perfect abstractions.

(For a larger system this whole process can be repeated on the module level)

Sverres Design Process

1. Brainstorm for Use Cases: Span functionality space; do not aim for "completeness".
2. Make design decisions: Divide into modules (++). (This is not a systematic process)
3. Map the Use Cases from 1 on the design. This is both a cosistency check and it yields the sub-use-cases for the modules.
4. Draw module interaction diagram. Who calls who?
5. For each module:
   - Sum up use-cases from 3.
   - Either: Design the perfect module interface that satisfies the use-cases - or recurse from 2.
6. Move responsibilities between modules (reorganize how the system is divided into modules if necessary) so that the diagram in 4. gets fewer arrows and that the module interfaces becomes perfect abstractions.

1

Error Dection: The learn-by-heart list

Error Dection: The learn-by-heart list

— Replication Checks
— Timing Checks
— Reversal Checks
— Coding Checks
— Reasonableness Checks
— Structural Checks
— Dynamic Reasonableness Checks

Note that enabling some of these may require extra code/insfrastructure.

1

Fault model and software fault masking

Learning Goals

Learning goals; Fault model and software fault masking

- Understanding of the three cases in low level design for fault tolerance by redundancy: Storage, Computation and Communication.
- Understanding of the work method: 1) Find error model 2) detect errors and merge failure modes (+error injection for testing) 3) handling/masking with redundancy ...aiming for progression of fail fast, reliable and available systems
- Ability to Implement (simple) Process Pairs-like systems.

1

Intro

What we have: Acceptance tests, Merging failure modes, Redundancy

So, given these elements: How would a fault tolerant system in fact be built? This chapter describes 3 examples to illustrate.

Chapter title:

Fault Model	Failure modes, probabilities and spec
Software Fault Masking	"Masking errors by redundancy"

Remember the Context here:

- To handle also unexpected errors -> Everything can fail -> Is everything hopeless?
- This kind of fault tolerance is a question of design
- But is it a global design? We would want composition: to be able to build fault tolerant systems from fault tolerant modules.
- This chapter: We are making *the* three basic modules (storage, communication and processing).

The underlying progression in the examples

- Failfast (all failures is detected immediately)
- Reliable (failfast + repair)

- Available (continous operation)

Rough Process:

1. Find the failure modes.
2. Detect errors / simplify error model.
   - injection of errors for testing
3. Error handling by using redundancy –> reliable & available module

Error injection:

- lets us test error handling!

Brainstorm: **How can we *test* the handling of unexpected errors?**

- Yes, (if acceptance tests are good) inject failed acceptance tests
- Btw: This is also simplified by merging of failure modes.

Case 1: Storage

Assume unreliable functions: read & write. We are writing sectors to the HD, but imagine an array of data areas.

Step 1) **Failure modes:**

Write	Read
Writes wrong data	Gives wrong data
Writes wrong place	Gives old data
Does not write	Gives data from wrong place
Fails	Fails

(Grays model also includes probabilities. -> We are reducing the chance of failure to less than some number here.)

Step 2) **Detection, Merging of error modes and error injection.**

- Writes also address, checksum, versionId, statusbit to the buffer. (version id isnt used yet, statusbit will be used for injection.)
- All errors -> fail
- Makes "decay" thread that runs in paralell and flips status bits (For testing).

Step 3) **Handling w. redundancy**

- More copies of buffer -> (The version id is used - The newest is returned)
- All reads implies writeback on error.

- Repair thread reading regularly.

Possible to calculate probabilities for error of the total system.

store_read

```
/* Reads a block from storage, performs acceptance test and returns status */
bool store_read(group, address,&value){
  int result = read(group,address,value);
  if(result != 0 ||
     checksum fails ||
     stored address does not correspond to addr ||
     statusBit is set){
    return False;
  }else{
    return True;
  }
}
```

1

The (error injection) Decay Thread

```
// There is one store_decay process for each store in the system
#define mttvf 7E5 // mean time (sec) to a page fail, a few days
#define mttsf 1E8 // mean time(sec) to disc fail is a few years
void store_decay(astore store){
  Ulong addr;
  Ulong page_fail = time() + mttvf*randf();
  Ulong store_fail = time() + mttsf*randf();
  while (TRUE){
    wait(min(page_fail,store_fail) - time());
    if(time() >= page_fail){
      addr = randf()*MAXSTORE;
      store.page[addr].status = FALSE;
      page_fail = time() - log(randf())*mttvf;
   }
   if (time() >= store_fail){
     store.status = FALSE;
     for (addr = 0; addr < MAXSTORE; addr++) store.page[addr].status = FALSE;
     store_fail = time() + log(randf())*mttsf;
    }
  }
}
```

1

Reliable Write

```
#define nplex 2  /* code works for n>2, but do duplex */

Boolean reliable_write(Ulong group, address addr, avalue value){
  Boolean  status = FALSE;

  for(int i = 0; i < nplex; i++ ){
    status = status ||
    store_write(stores[group*nplex+i],addr,value);
  }
  return status;
}
```

1

reliable_read

```
bool reliable_read(group,addr,&value){
  bool status, gotone = False, bad = False;
  Value next;

  for(int i = 0; i < nplex; i++ ){
    status = store_read(stores[group*nplex+i],addr,next);
    if (! status ){
      bad = True;
    }else{       /* we have a good read */
      if(! gotone){
        *value = next;
        gotone = TRUE;
      } else if (next.version != value->version){
        bad = TRUE;
        if (next.version > value->version)
          *value = next;
      }
    }
  }
  if (! gotone) return FALSE;   /* disaster, no good pages  */
  if (bad) reliable_write(group,addr,value); /* repair any bad pages  */
  return TRUE;
```

1

This **can** still fail silently! Under which circumastances?

The Repair Process

```
void store_repair(Ulong group){
  int i;
  avalue value;
  while(TRUE){
    for (i = 0; i <MAXSTORE; i++){
      wait(1);
      reliable_read(group,i,value);
    }
  }
}
```

1

(These code snippets are taken from some presentation helt by Gray found on the internet.)

Case 2: Messages

Fault tolerant communication demands storage & processes in order.

Step 1) Failure modes Message: Lost, delayed, corrupted, duplicated, wrong recipient.

Step 2) Detection, Merging of error modes

- Session Id
- checksum
- ack
- sequence numbers
- All errors -> Lost message

Step 3) Handling w. redundancy

- Timeout & Retransmission.

Case 3: Processes/Calculations

Note: Does not require safe communication

Step 1) **Error mode**: Does not do the next correct side effect.

Step 2) **Detect&Simplify**: All failed acceptance tests -> PANIC/STOP (Failfast)

Step 3) **Handling w. redundancy**: 3 solutions:

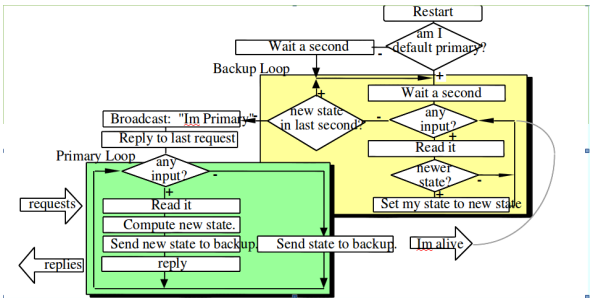
Alt 1: 3a) **Checkpoint-Restart** (Reliable)

- Writes state to storage after each acceptance test, before each side effect
  - (Note this very important pattern.)
  - (Yes, we are very dependant on good acceptance tests!)
- This gives us error containment.

Alt 2: 3b) **Process pairs** (Gives us available)

- Two processes, Primary & Backup (primary does the work)
- Backup takes over then the primary fails
- IAmAlive messages from primary to backup
- Primary sends checkpoints to backup.

Process Pairs



1

process pairs pseudocode

```
repeat
  // Backup mode
  Read Checkpoint and IAmAlive messages
  Update state
Until last IAmAlive is too old

Broadcast: IAmPrimary

Finish active job  // Possibly a duplicate

repeat
  // Primary mode
  if new request/task in job queue then // Part of the state
    do work
    if acceptance test fails then
      restart.
    else
      send checkpoint & IAmAlive // Unsafe communication!
    answer request/commit work.
  else
    if it is time to send then
      // The Assumption is that there will be more of these.
      send last Checkpoint & IAmAlive.
```

1

NB! **This does not rely on safe communication**

- Redundancy by resending: Masks communication errors.
- Note new master can answer request sent to old master (no sessions)
- (Communication queues part of state).
- ...

Alt 3 3c) **Persistent Processes**

- Assumes a transactional infrastructure.
- All calculations are transactions ("atomic transformations from one consistent state to another").
- The processes becomes stateless - All states are in database
- For such simple processes the OS can take care of take-over & restart!

In grays vision: **This** is the context of databases!

- But of course: Reliable/available Storage/Communication/Calculations is necessary for making this transactional infrastructure.
- So for us, where it is not, we need to do Alt 1 or 2.

Project Design Evaluation

Specification

You are going to present your design of the lift system. The main point is to decide, and be able argue clearly, how you will solve the fault tolerance challenge ("No orders should ever be lost"). The design should be followed through at least until the system is divided into modules and the module interfaces are complete, realistic and relatively stable.

Evaluation Criteria

- Solution: Did you solve the problem? Are we confident that your system will handle the plthora of errors and still handle all orders?
- Modules: Are the modules you have chosen well named, and gives a reasonable partitioning of the system (do we get the feeling that dependencies/coupling between the modules are minimized and that everything is thought through?)

- Module interfaces: Do the module interfaces seem nice abstractions of the modules responsibilities?
- Presentation: Did you manage to convey your solution and your modules clearly?

The way you describe the module interface will be different depending on how modules interact/communicate. Ex. In C a module is typically described by its header-file, while in Go a list of services/incoming messagetypes that a server should handle would typically describe its interface.

Presentation

We will spend 15 minutes per group (a hard maximum). Prepare a presentation aiming for 10 minutes, presenting your design:

- How you solve the fault tolerance challenge.
- Which modules you have chosen to partition the system into. (Including "interfaces").
- Go through a tricky scenario or two?

The SW design process

Many thick books has been written about systematic approaches to software design...

Here is my take:

1. Have a brainstorm on "what the system must handle": You must handle "that a button is pressed", "that a node is killed", etc. This need not be a complete set of scenarios, but should be illustrating different aspects of the system functionality. (I would guess 4-8 scenarios should span the functionality space)
2. Divide the system into modules based on your gut feeling and discussions with your partners.
3. Map the scenarios onto the modules in a "then this must happen" manner: "The button press is detected by the X module, that notifies the event loop in the Y module. Then a message is sent ... etc." (Also here: do not give in to the temptation to make "complete" scenarios, focus on spanning the space.)
4. Draw the diagram over module interactions. Who calls who?
5. Sum up, for each module, all its incomming interactions/requests. These are the specification *of the module*.

6. Try to make perfect module interfaces which fulfils each module's responsibilities.

7. Move responsibilities between modules (reorganize how the system is divided into modules if necessary) so that the diagram in 4. gets fewer arrows and that the module interfaces in 6. becomes perfect abstractions.

For a larger system this whole process should be repeated on the module level.

Project Design Evaluation

Show documents

— designevaluation.pdf (by Sverre)
— design_scoresheet.pdf (used v2018)

Slides summing up Sverres project design

Big Scenarios/Use Cases

Big Scenarios/Use Cases

- Button
- Floor
- A node stops/crashes/disappears
- A node starts
- A node looses network
- A node gets network

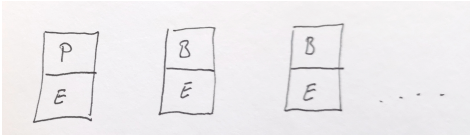
1

Division into 'modules' - 'Responsibilities'

Division into 'modules' - 'Responsibilities'

(Step II: Gut feeling decitions!)
Responsibilities:

- Local elevator and HW
- Decitionmaking (in Primary)
- System Configuration



Decitions:

- Very thin Elevator; All decitions made by Primary

1

Scennario: Node Starts

Scennario: Node Starts

A node starts:

- Listens for 5 secs for a Primary broadcast
- Got one: I am backup: continue listening for Primary broadcasts and updating state.
- If not: I am primary: Start broadcasting Primary broadcasts. Start listening for new order messages.
- Start elevator-control handling orders and setting lamps and checking buttons and floor sensor.

Preliminary Decitions (to be checked/confirmed with other scenarios):

- The Primary broadcasts state at updates and regularly

1

Button

A button is pressed

A button is pressed:

- Register UnconfirmedOrder; Send unconfirmed-orders to Primary.
- On Primary; Make decition, update state, send Primary broadcast.
- All Elevators: Set relevant new lamps.
- Originating Elevator: Check lamps to remove orders from UnconfirmedOrder set.

Decitions:

- The Unconfirmed-orders are sent, at need and repeatedly, even when empty, to Primary (UDP)
- UnconfirmedOrder module is roughly ready: Add(order), Remove(lamps) and Tick().

1

Terms and techniques

Terms and techniques, used

- Fault Tolerance (The term)
- Error modes (TCP vs UDP when loosing network)
- Static redundancy (UDP and broadcast resending, storing state in all nodes.)
- Dynamic redundancy (Take-over of orders when an elevator crashes)
- Error detection (Timeouts, Reentering too old orders)
- Backward Error Recovery (Restart goes back to safe state)
- Forward Error Recovery (Redistribution of orders)
- Acceptance Tests
- Merging of error modes (Any error leads to restart, timeout is caused by more reasons)
- Writing checkpoints, checkpoint-restart (Getting state from Primary at startup)

1

What now?

What now?

1

Atomic Actions and Transactions

Learning Goals

Learning Goals: Atomic Actions

— A thorough understanding of the problems Atomic Actions are meant to solve and how these motivates the different aspects of Atomic Actions.
— Ability to use and implement Atomic Actions, including the mechanisms providing the start, side and end boundaries.
— Understanding the motivation for using Asynchronous Notification in Atomic Actions
— A coarse knowledge of how the mechanisms for Asynchronous Notification in C/Posix, ADA and Java works.

1

Learning goals; Transaction Fundamentals

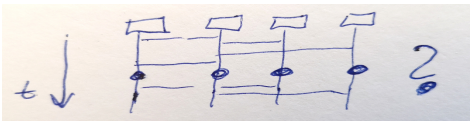
— Knowledge of eight “design patterns” (Locking, Two-Phase Commit, Transaction Manager, Resource Manager,Log,Checkpoints,Log Manager, Lock Manager), how they work and which problems they solve. Ability to utilize these patterns in highlevel design.
— Comprehension of terms: Optimistic Concurrency Control, Two-phase commit optimizations, Heuristic Transactions, Interposition.

1

The problem of syncing recovery points

Syncing recovery points: The problem

So; We are to sync recovery points to avoid the domino effect.



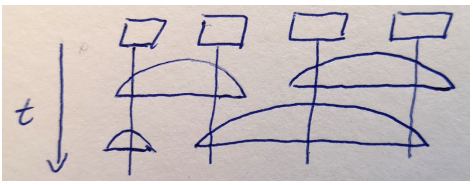
— Recovery must still be written after acceptance tests - when we know everything is in order.
— But looking more closely at the recovery point: It represents the consistent state *at the start* of an operation, while the acceptance tests are run at the end.
— When do we sync? Does such a time exist where all actors, for all operations reaches a sync point?

1

The solution

The Solution: Syncing Recovery Points

Basic Design Infrastructure; Three 'Boundaries':



— Start Boundary: When an actor starts participation in an operation: (Register membership. If preparing for backwards error recovery; store recoverypoint of relevant state subset?)
— Side Boundary: No interaction with actors not participants of the operation. (No communication, 'Lock' any relevant state subset to operation.)

1

How can this be implemented?

How to implement the AA infrastructure?

So, AA implementation?

1. Threads, interacting with each other? (like, in the domino-figure)
   - Participants supporting protocols
   - Go example?
   - Ada 'Task'
2. Server-processes that handles requests/'supporting protocols'? (same, just better structured: like in Gray)
3. A number of procedures that is called by participating threads? (the B&W book)
   - Action == Class or Package/module?
   - Thread interaction through synchronization and shared variables.
4. Threads that is created and terminated with the AA? (Cool, *the* way to get forward recovery in C)
   - select-then-abort in Ada
   - Asynch. exceptions in Java
   - pthread_cancel in POSIX

1

Example slides from B&W

Show Burns & Wellings slides on AA

Ada: Relevant slides from B&W: Chap. 7: 10-16
Java: Chapter 10 (ThirdEdition of the books slides) Slide 20-26
Comments on examples:

- Start Boundary ok
- Nice (?) Action == Class/Object/Package
- No error handling
- No side boundary
- static participants & actions.
- How is one of these actions initiated?
- reusable functionality in controller: membership, voting

1

Sidestep: java monitors

Sidestep: Thread synchronization in Java

Java Monitors:

- **Synchronized** methods:
  - Only one thread can call one of those in an object at the same time. (need object lock) (Btw. Classes have locks also).
  - Almost like if all methods reserved a mutex when running.
- wait(): The thread is suspended and the lock released.
- notify()/notifyAll(): Awakens one/all threads (but they still need the lock, and cpu, to continue).

1

Sidestep: ADA Protected Objects

Sidestep: Thread synchronization in Ada

ADA: Protected Objects:

- Modules (as with Monitors)
- Functions (readlocks)
- Procedures (writelocks)
- Entries w. Guards (not CondVar)
- Blocking from the inside is an error! (Waiting on other monitor is not blocking Waiting on guard or on task entry is...)
- Guards tests only private variables. -> reevaluation only when exiting entry&proc.

Term: The 'Guard':

- A test integrated in the language that opens or closes for execution
- In our context: blocking, suspending.
- Is not (necessarily only) evaluated sequentially

1

4 discussion points:

Four discussion points

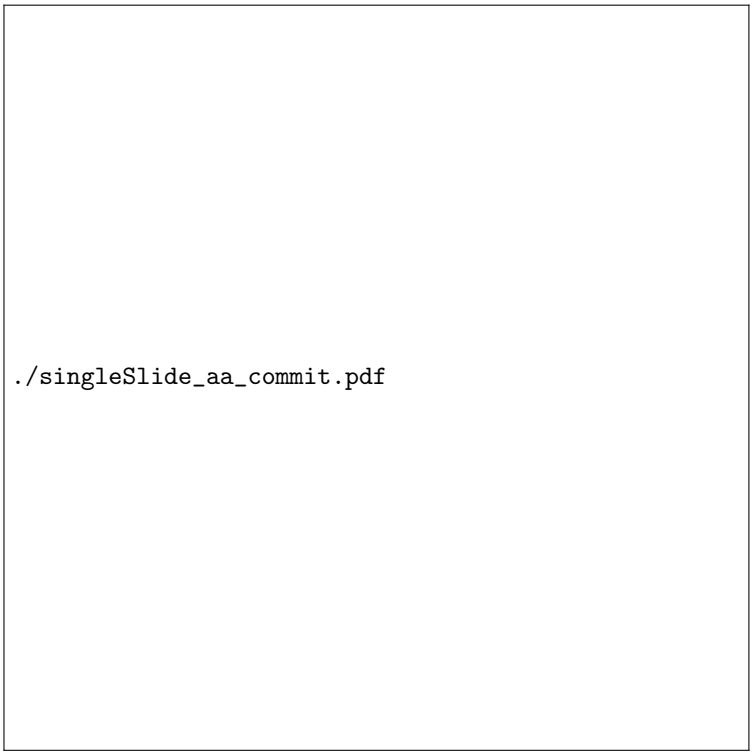
1. Locking (one form of non-interaction) - Two phases: Growing and Shrinking
   - (The other form of non-interaction is not sending messages :-))
2. How to sync at the end: The (two-phase) 'commit protocol'
   - (Well, a 'barrier' would do it, if no error handling)
3. The 'Log' and 'Checkpoints' — An intelligent replacement for the recoverypoint.
4. How to distribrute error information.
   - Via the commit protocol.
   - Some polling regime (shared variables or messages).
   - *Asynchrouneous transfer of control*

1

Locking

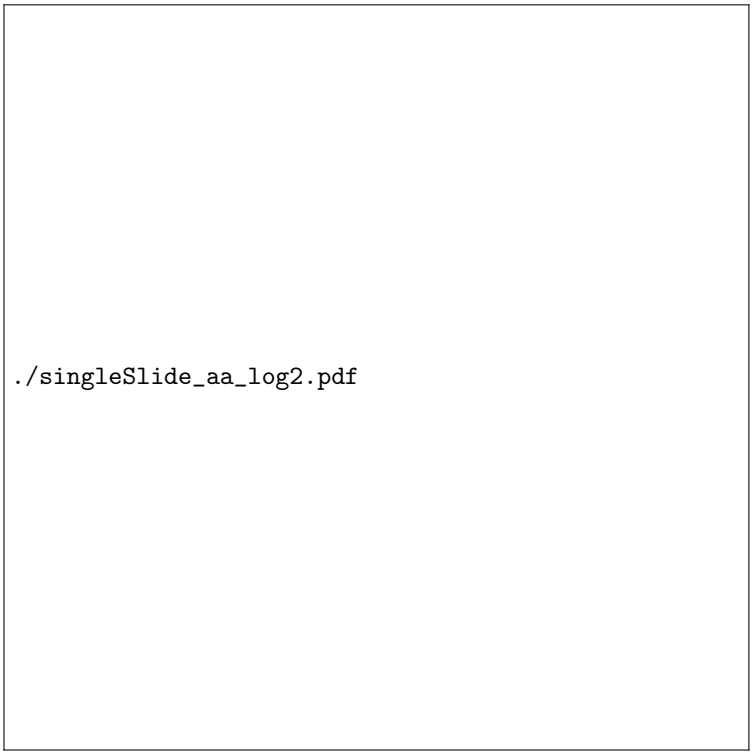
1

Syncing acceptance tests



Class challenge: do we really ned to lock when doing a read-only operation?

Log and checkpoints



How to distribrute error information.

./singleSlide_aa_error.pdf

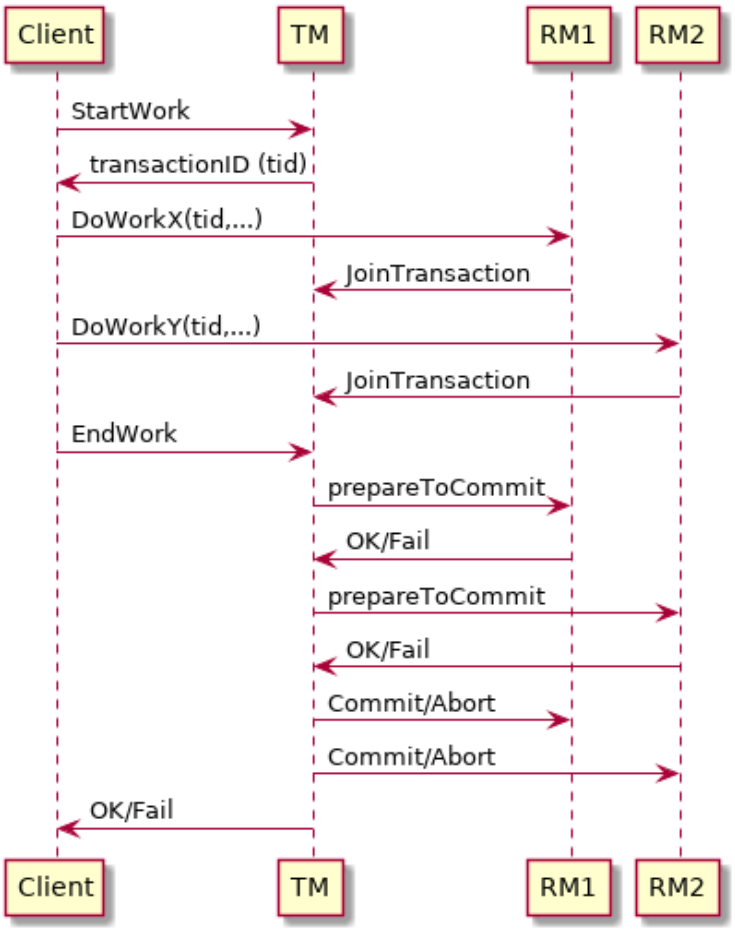
Four implementational patterns

- The 'Transaction Manager' or 'Action Controller'.
- The actor: 'Resource Manager', thread, or 'action member'.
- The 'Lock Manager'.
- The 'Log Manager'.

1

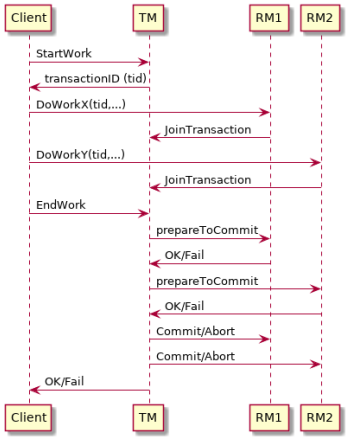
Sequence diagram for a transaction.

As a summary: Go through sequence diagram for transactions



Comment on Locking also, to explain side boundary.

A Transaction



Some error cases

1

End

Template

SlideTemplate

1