

# Performance Patterns, Pitfalls, and Approach

by *Gerrit Muller* HSN-NISE

e-mail: `gaudisite@gmail.com`

`www.gaudisite.nl`

## Abstract

Performance Design is based on the application on many performance oriented patterns. Patterns are a way are to consolidate experience: what solution fits to what problem in what situation? Pitfalls are also a way to consolidate experience: what are common design mistakes?

October 20, 2017  
status: preliminary  
draft  
version: 0.1

## *Common Platforms and Bloating*

Generic nature of platforms

Most SW implementations are way too big

Performance suffers from oversize and generic provisions

# Exploring Bloating: Main Causes

>90% of all Software statements are not needed, but caused by:

over-specification

bad design

too generic

dogmatic rules

legacy remains

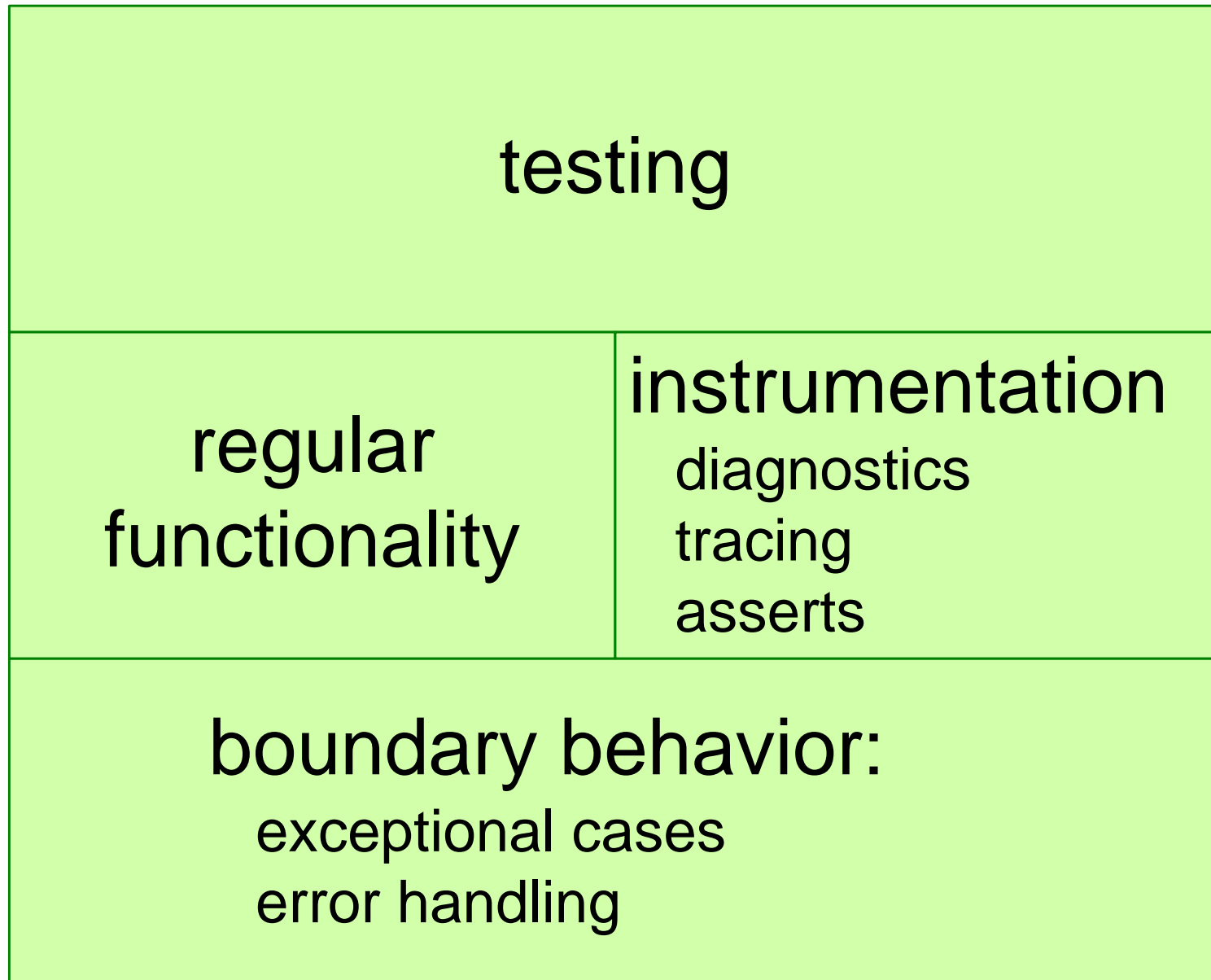
core function  
less than 10%

legend

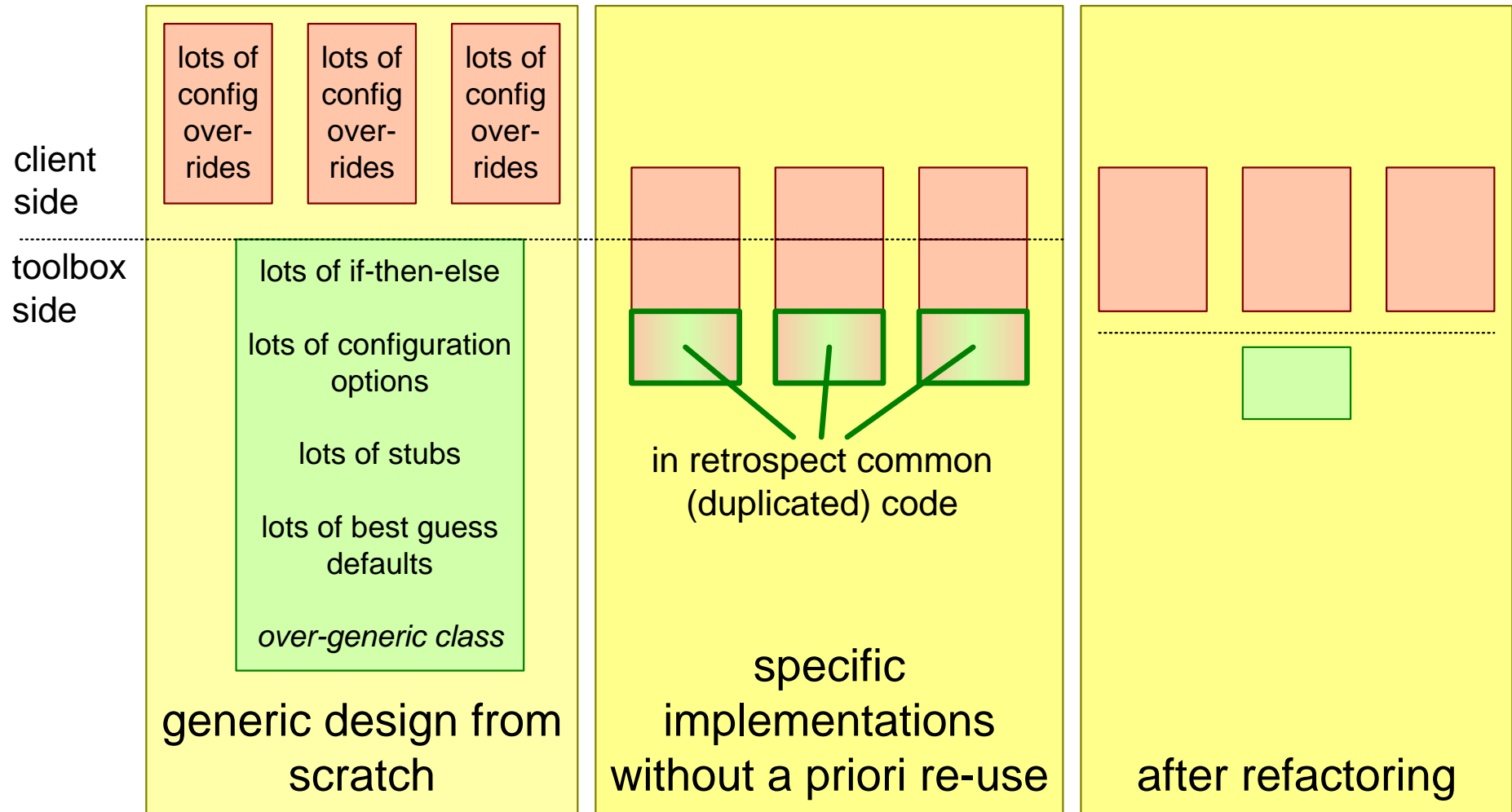
overhead

value

# Necessary Functionality $\gg$ Intended Regular Function

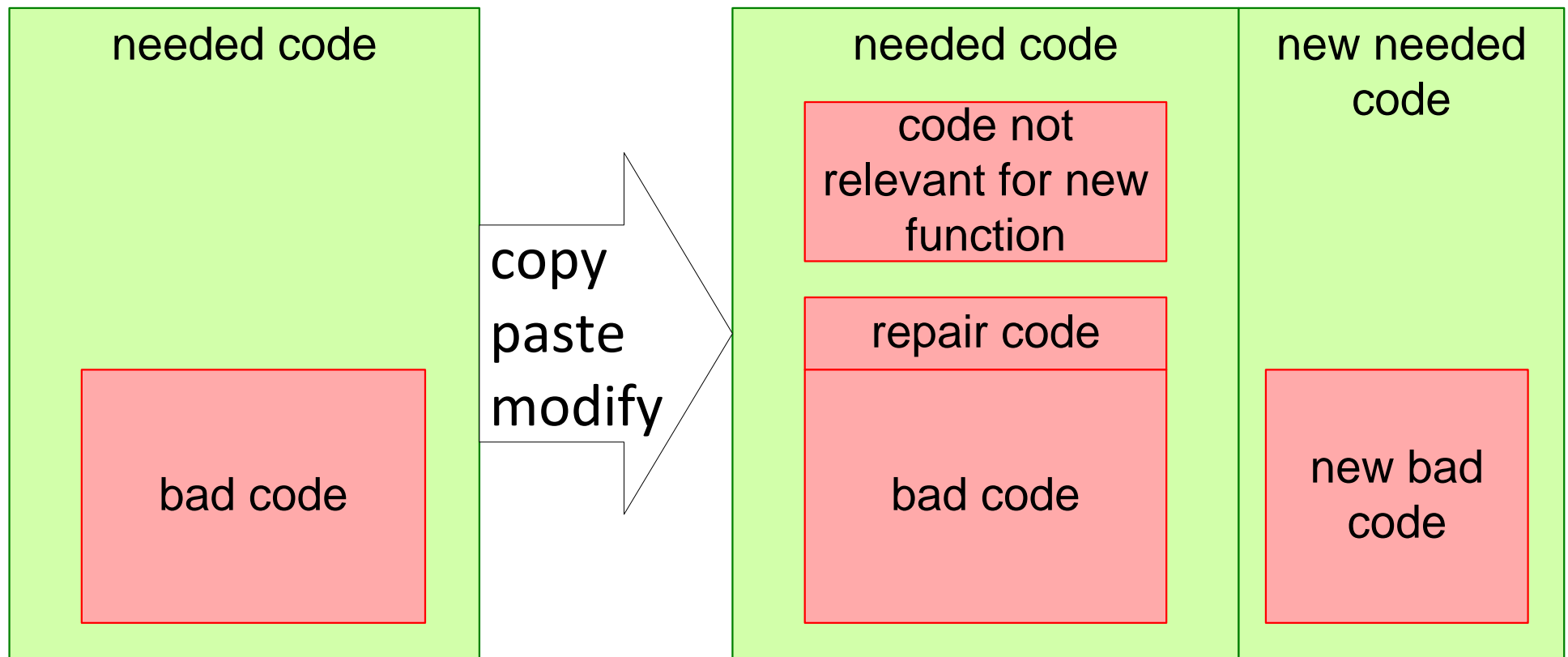


# The Danger of Being Generic: Bloating

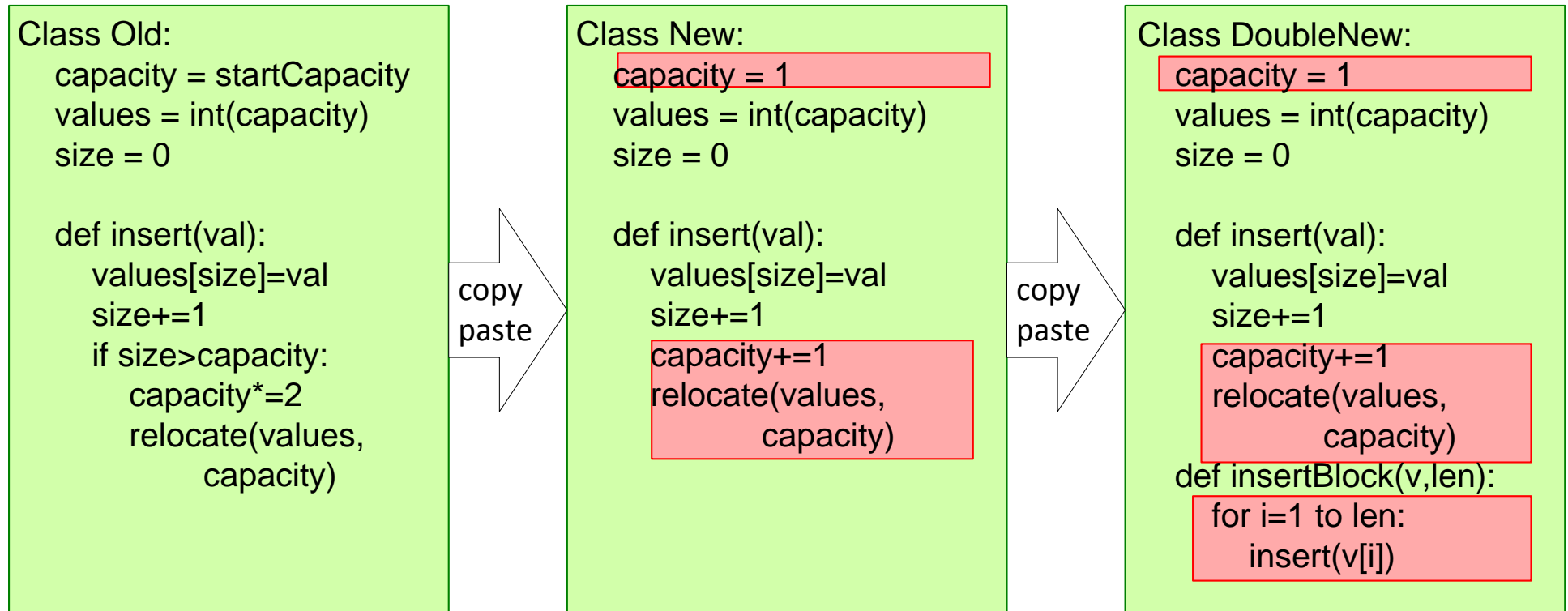


"Real-life" example: redesigned *Tool* super-class and descendants, ca 1994

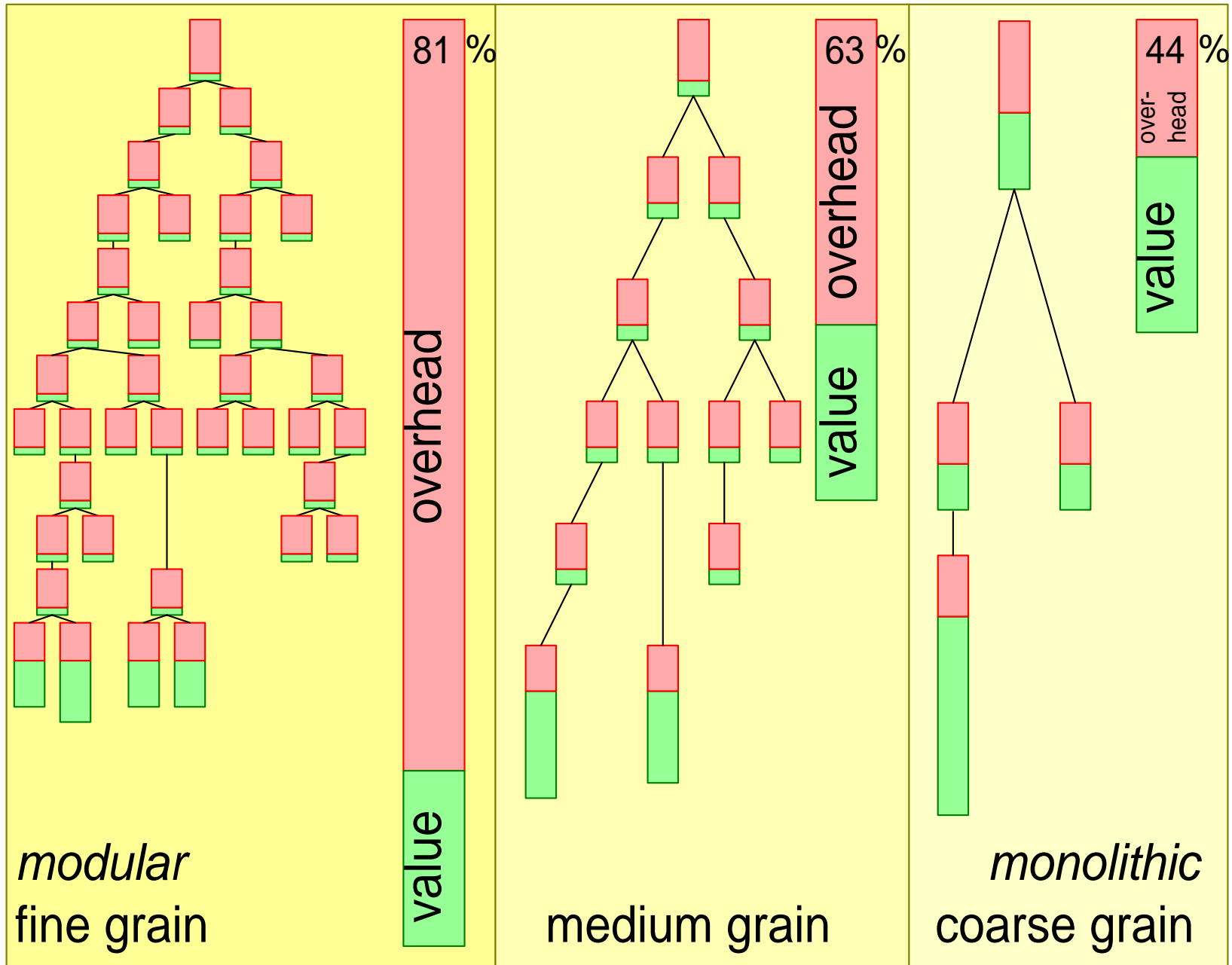
# Problem Propagation via Copy & Paste



# Example of Problem Propagation

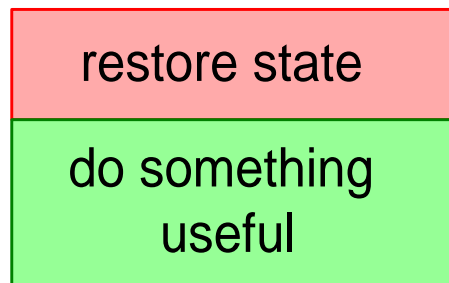
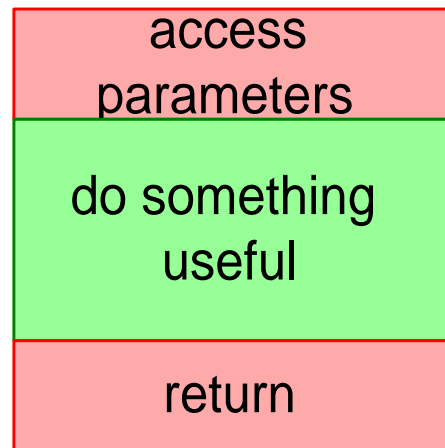
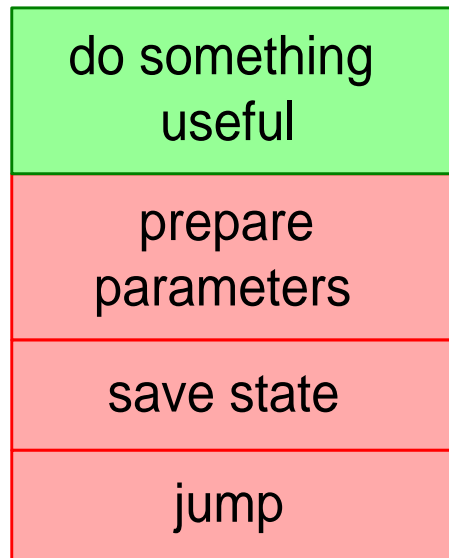


# Overhead Penalty of Modularity





# Function Call Overhead



Load and depth dependent  
(hidden) side effects

pipeline flush  
I-cache disturbance  
D-cache disturbance

legenda

overhead

value

Suppose:

Call Overhead =  $10\mu\text{s}$

Call graph branching factor = 2

Depth = 12

What is the Call overhead  
when all branches are followed?

Suppose:

Function call =  $10\mu\text{s}$

Call layer depth = 20

1024 calls per image

What is the maximum frame rate possible assuming that the complete CPU time is available for function calls?

## *Common Platforms and Bloating*

Platforms are overprovisioned and very generic

Are benefits > disadvantages?

Performance loss is significant and can be measured and modelled

## Multi-Dimensional Viewing of many Images: Greedy and Lazy Design Patterns

## Greedy and Lazy systems



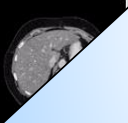




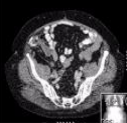
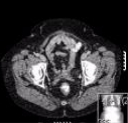
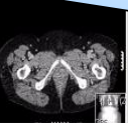
Greedy: pre-fetched lots of data:

System tries to have data available for the requesting system

Lazy: hardly of no pre-fetching of data:

System tries to set data available for the requesting system  
only when asked for

## Example Greedy / Lazy (1)

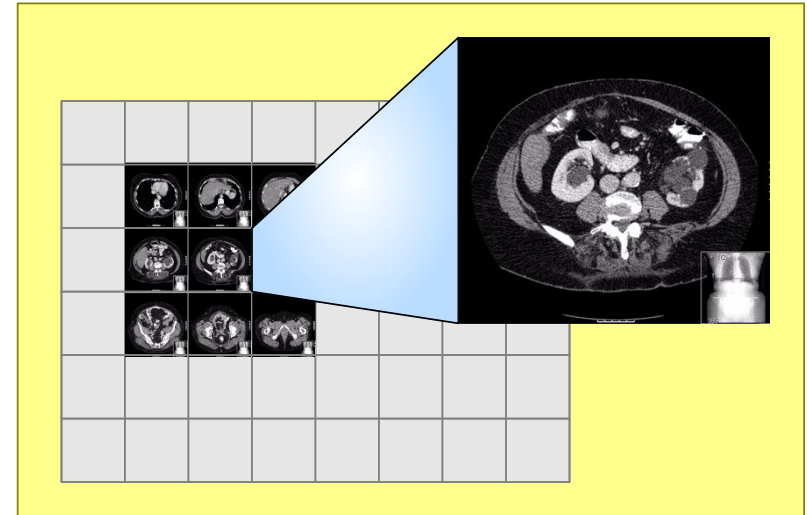
META DATA	META DATA	META DATA	META DATA	META DATA	META DATA	META DATA	META DATA
META DATA							
META DATA							
META DATA							
META DATA	META DATA	META DATA	META DATA				
META DATA	META DATA	META DATA	META DATA	META DATA	META DATA	META DATA	META DATA
META DATA	META DATA	META DATA	META DATA	META DATA	META DATA	META DATA	META DATA

META DATA=  
Patient name  
Slice nr. / position  
annotation  
explanation  
date / time

## Example Greedy / Lazy (2)

Lazy: Fetch only  
the requested image

Greedy: Fetch all the images  
in the set



In between options:

- Fetch requested image + surrounding images
- Fetch requested image + only meta information of images



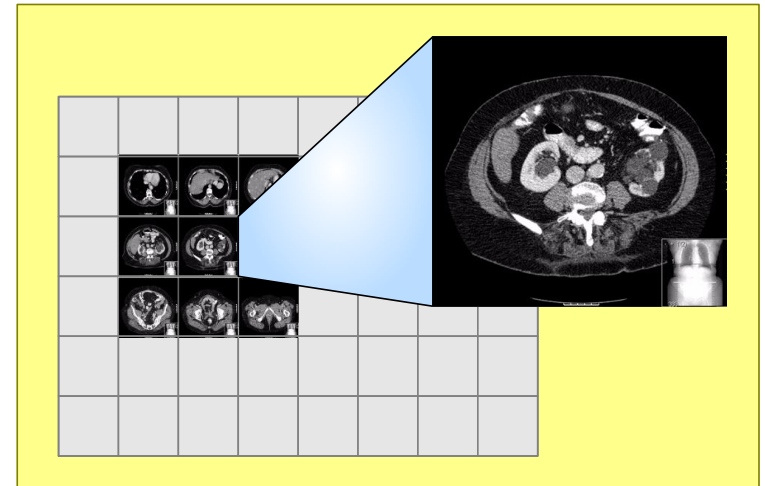
## Example Greedy / Lazy (3)

### Lazy:

- low load on system
- long waiting time for next image

### Greedy:

- high load on system
- possible long initial wait
- short response time in steady state

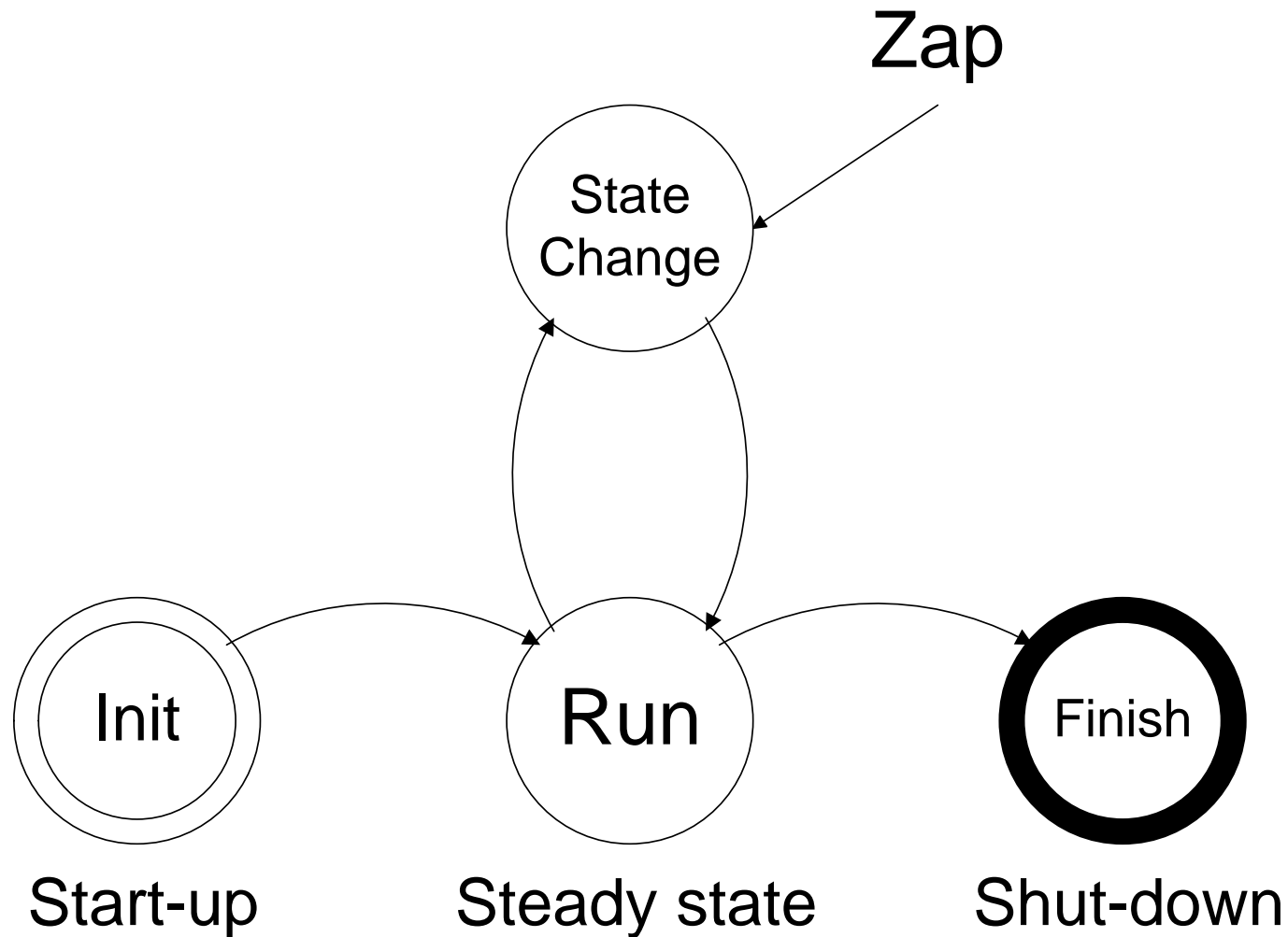


### In between options:

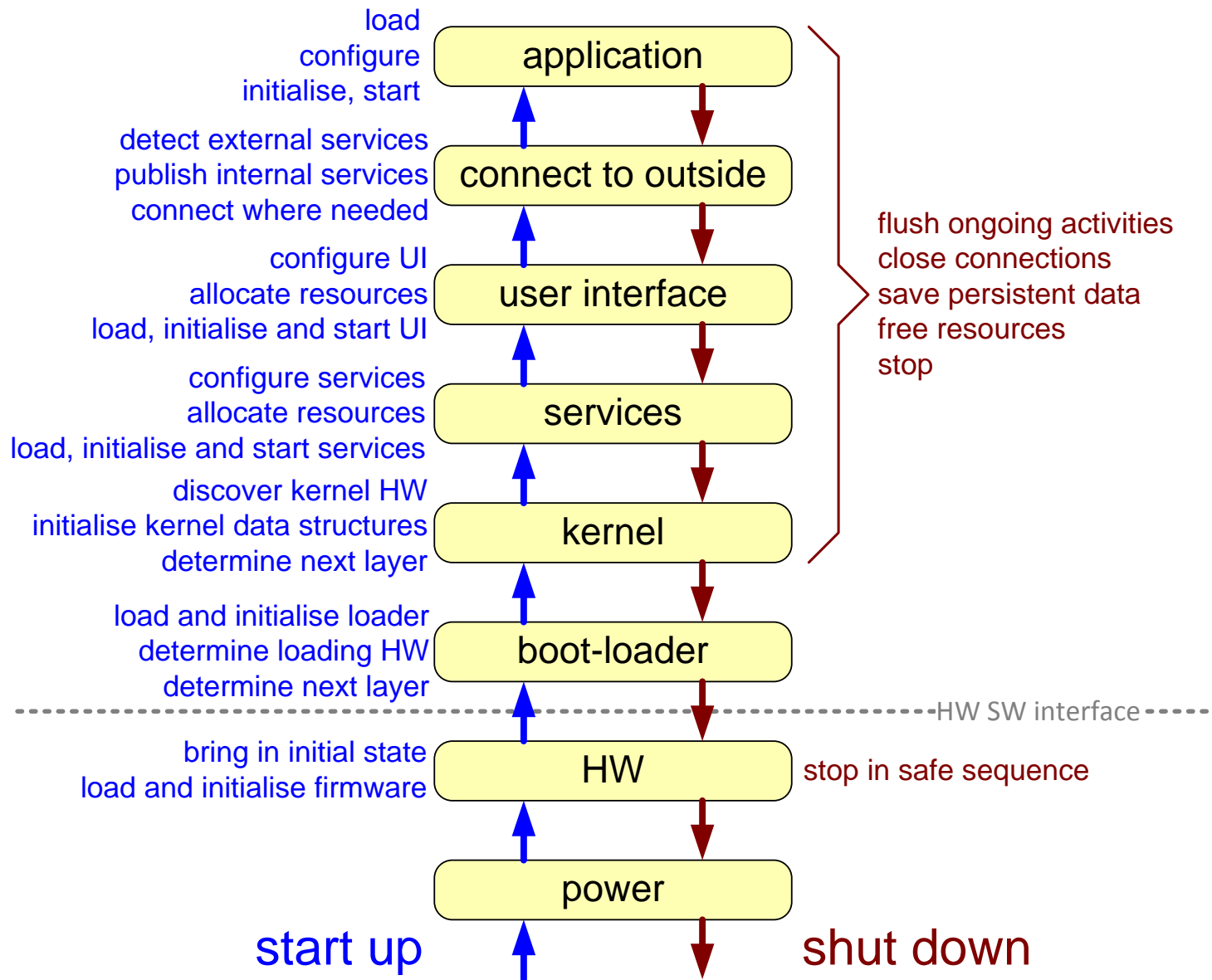
- medium system load
- fast response for initialization and common image fetches

## Initialization, Steady State and Finalization

# Start-up, Steady State, Shut Down



# Start-up, Steady State, Shut Down Scheme



# Start-up, Steady State, Shut Down Trade off

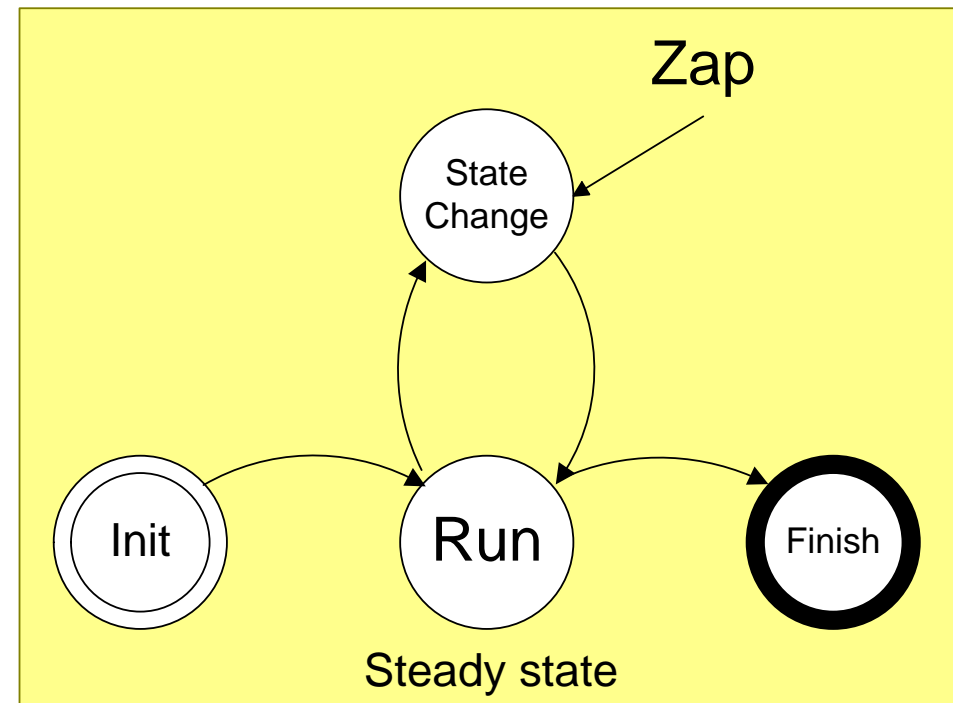
Trade-off:

**Optimize on steady state**

may result in  
poor performance for initialization  
and process finish

**Optimize on Initialization  
and/or finish**

may result in  
poor steady state performance



# Common Performance Pitfalls

---

- Overhead
- Data bloating
- Cache thrashing
- Layering
- Process communication
- Conversions
- Serialization
- Backfiring optimisations
- Hidden loads (bus, DMA etc)
- Poor algorithms
- Wrong dimensioning

The ASP™ course is partially derived from the EXARCH course developed at *Philips CTT* by *Ton Kostelijk* and *Gerrit Muller*.

Extensions and additional slides have been developed at *ESI* by *Teun Hendriks*, *Roland Mathijssen* and *Gerrit Muller*.