Arsitektur Perangkat Lunak

Semester Ganjil 2023/2024

Kurnia Saputra, ST, M.Sc.
kurnia.saputra@usk.ac.id

Jurusan Informatika
Fakultas Matematika dan Ilmu Pengetahuan Alam
Universitas Syiah Kuala

Kontrak Perkuliahan

- Jadwal
  - **Kuliah**: Kamis, pukul 08.15 – 09.55 WIB.
  - **Praktikum**: mulai minggu ketiga perkuliahan, jadwal menyusul.
- Materi kuliah dan praktikum didistribusikan melalui laman elearning Unsyiah

  `http://elearning.unsyiah.ac.id/` (check regularly!)
- Bobot penilaian: tugas **20%**, praktikum **10%**, UTS **35%**, UAS **35%**

Prasyarat: memahami konsep dasar rekayasa perangkat lunak; memahami konsep UML; tidak perlu mahir bahasa pemrograman!

Bidang Minat Rekayasa Perangkat Lunak

Semester Ganjil

- INF525 Rekayasa Kebutuhan Perangkat Lunak
- INF527 Arsitektur Perangkat Lunak

Semester Genap

- INF518 Proyek Perangkat Lunak
- INF526 Perangkat Lunak Berbasis Komponen
- INF528 Kualitas Perangkat Lunak

Materi Kuliah

- Patterns
  - Problem Frames (analysis)
  - Analysis Patterns
  - Architectural Patterns (coarse-grained design)
  - Design Patterns (fine-grained design)
  - Test Patterns
- Architecture
  - Life Cycle
  - Industry and Business
  - In the Cloud
- Components (MK Perangkat Lunak Berbasis Komponen)
  - Component definition and specification
  - Component models
    - Java Beans
    - OSGi
  - Component-based software development process

Jadwal Perkuliahan

Pertemuan	Topik
1	Pengenalan Arsitektur Perangkat Lunak - Definisi, Penelitian terkait, ADIT Development Process
2	Basic Terminology - Requirements, Facts, Assumptions, Phenomena, Domain Knowledge
3	Problem Frames
4	Analysis Patterns - Planning
5	Analysis Patterns (lanjutan) - Trading
6	Architectural Patterns - Definisi APL, Why Architecture?, Module Pattern, Component & Connector Patterns
7	Architectural Patterns (lanjutan) - Client-Server Pattern, Peer-to-Peer Pattern, SOA Pattern, Publish-Subscribe Pattern, Shared-Data Pattern
8	UTS

Textbooks

Textbooks:

- Design Patterns: Elements of Reusable Object-Oriented Software – Gamma, Helm, Johnson, and Vlissides (1995)
- Software Architecture in Practice – Bass, Clements, and Kazman (2012)
- Problem Frames – Jackson (2001)

Your Expectations

- Mengapa mengambil mata kuliah ini?
- Apa harapan Anda dari mata kuliah ini?

Your definition of Software Engineering

Discuss in a group of 2 and write your definition down.

Software Engineering: definition

Software Engineering $\neq$ Programming

Software Engineering (Balzert):

Goal-oriented provision and systematic use of principles, methods, concepts, notations and tools for team-based development and application of large software systems according to engineering principles. Goal-oriented means e.g. taking costs, time and quality into account.

Software system

A system, whose system components and system elements consist of software.

Software: program + documentation

(Later on, we will use the term "machine" for software systems.)

Why do mere programming skills not suffice?

- (Practically) **all** software contains defects.
- This leads to an immense economic loss and the endangering of human life.
- Why is that so?
- What is new promising areas of research?

From CACM, Sept. 2008, p.55

...software has been exhibiting declining levels of dependability. This can be attributed to many factors, including

- A naive belief that anyone who can write software can write *good* software.
- A mistaken belief that running a few representative test cases indicates that the software is "correct" or adequate.
- Failure to understand that realizing a good *design* is more important than producing vast quantities of code and that the goal of software engineering is not only to produce code but also to produce trustworthy solutions to problems that can eventually be implemented in a programming language.
- Failure to realize that making changes to software – and in particular unnecessary, uncontrolled, and careless changes – can have an effect on its appropriateness and validity ...

Studies of the Standish-Group (CHAOS Research) I

http://www.standishgroup.com/sample_research

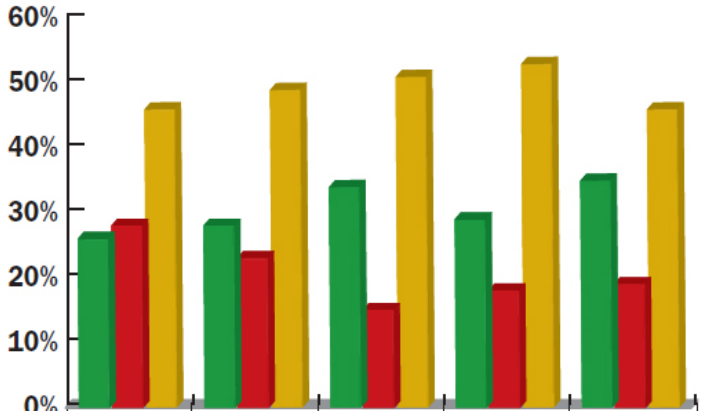
Alfred Spector

Bridges are normally built one-time, on-budget, and do not fall down. On the other hand, software never comes in on-time or on-budget. In addition, it always break down.

Studies of the Standish-Group (CHAOS Research) II

1994	1996
16%	27%
53%	33%
31%	40%

CHAOS PROJECT RESOLUTION



	1998	2000	2002	2004	2006
Succeeded	26%	28%	34%	29%	35%
Failed	28%	23%	15%	18%	19%
Challenged	46%	49%	51%	53%	46%

The above chart shows the results of project resolution over the last decade. This data is from our CHAOS Research project on project success and failure and covers more than 60,000 projects.

2009
32%
24%
44%

Study of the Research Triangle Institute on behalf of the National Institute of Standards and Technology, May 2002 I

- ⇒ Transportation equipment manufacturing sector: Each year, users detect on average 40 major errors and 70 minor errors.
- ⇒ Finance services error: Each year, users detect on average 40 major errors and 49 minor errors.
- ⇒ Estimated, projected costs of the US-economy per year: 59.9 bn. Dollar.

Study of the Research Triangle Institute on behalf of the National Institute of Standards and Technology, May 2002 II

"Software developers already spend approximately 80% of development costs on identifying and correcting defects, yet few products of any type other than software are shipped with such high levels of errors."

"If Microsoft made cars instead of computer programs, product-liability-suits might now have driven them out of business."

Spectacular Failures

- ⇒ Radiation device Therac-25
  dead and injured through overdose of radiation
- ⇒ Crash of the Ariane 5 launcher
  loss: 500 mill. Dollar
- ⇒ Luggage conveyor at the new airport in Denver
  costs: $1,1 mill. per day
- ⇒ Other examples are mentioned almost every month in the newspaper! (E.g., toll collect, Hartz IV, ...)

More information:
Peter Neumann: Computer-Related Risks, ACM Press 1995
News-Group *comp.risk*

What do we learn out of this?

- ⇒ The reuse of software generally is a good idea, but profoundly nontrivial.
- ⇒ In case of change in requirements, the software also needs to be changed.
- ⇒ Software must be specified so that it is comprehensible and can be used appropriately.
- ⇒ Quality assurance measures, such as tests and reviews are very important and should not fall victim to economic constraints.
- ⇒ Software should be robust and fault-tolerant (and a good deal more).
- ⇒ No useless function should be executed, especially not for historical reasons.

The field of software engineering deals with all these types of subjects!

Why can't software be built in the same way as cars and houses?

Software is something special, because it

- is intangible
- does not wear off through use, but ages because of changes or and no changes
- is not restricted through physical laws
- is easily alterable
- is difficult to measure, i.e. describe in a quantitative way
- does not exhibit a continuous but a discrete behaviour (no safety margins possible, small causes can have great effects.)

Therefore, we need specific methods for constructing software!

Summary

- Computer Science (and thus software engineering) is a very young science.
- Only small part of software development is programming.
- Due to the special features of software, specific engineering methods are necessary, but have not reached maturity yet.
- An important goal is to develop software with fewer faults.
- For this, promising and exciting new approaches exist.

Recent Developments: From "Art" to "Engineering"

Which steps lead from "Art" to "Engineering"? I

- **Model-based development**
  - develop sequence of models, each describing different aspects of the system/machine
  - models can be analysed and checked for coherence
- **Object Orientation**
  - Software architecture follows data, not functionality
  - Software as dynamic collection of communicating objects
  - Improved reusability through encapsulation of data
- **Patterns**
  - Templates for the different artifacts generated in software development
  - Useful in all phases of software development
  - Reuse through instantiation

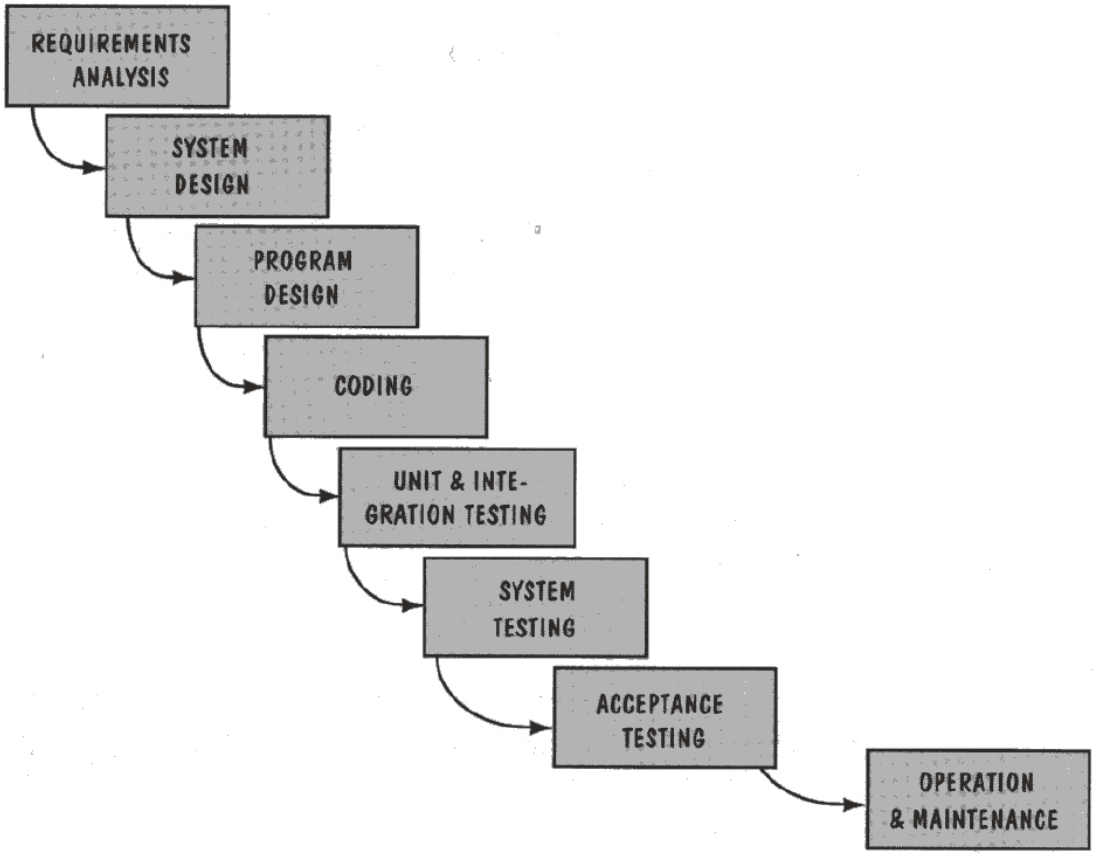
Which steps lead from "Art" to "Engineering"? II

- **Component Software**
  - Build software system from ready-made parts
- **Aspect-oriented programming**
  - Write different programs, each covering different aspects (e.g. computation vs. graphical representation) of the software
  - Compilers combine the different aspect programs to one executable program
- **Software engineering for special applications**

  e.g. Internet and multimedia applications

Software Process Models

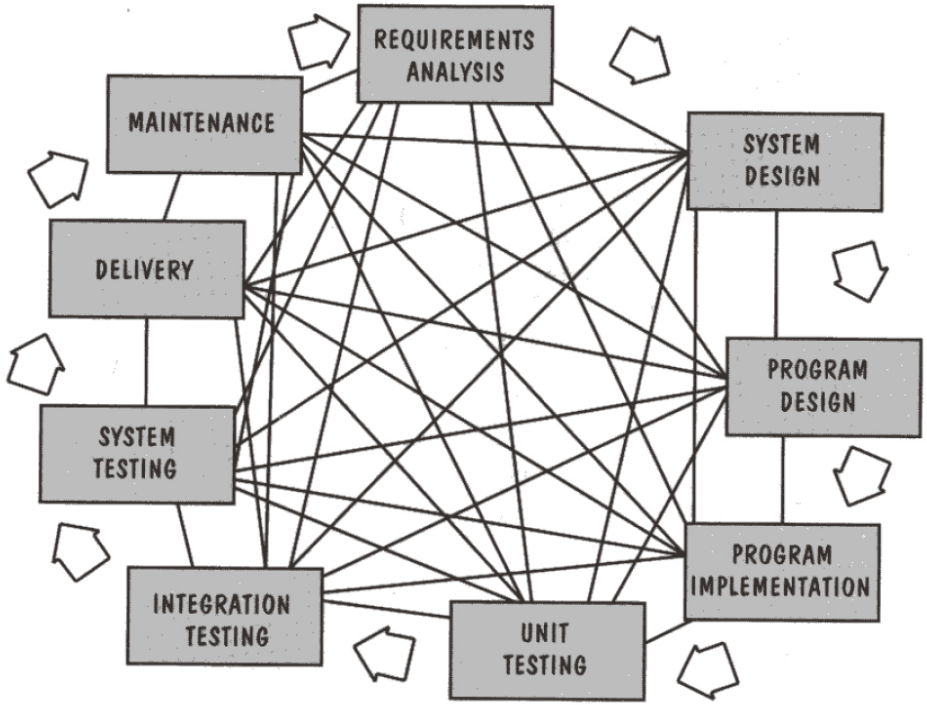
The Waterfall Model I



Idea: each phase results in a document which serves as initial point of the next phase. No return to earlier phases[1].

The Waterfall Model II

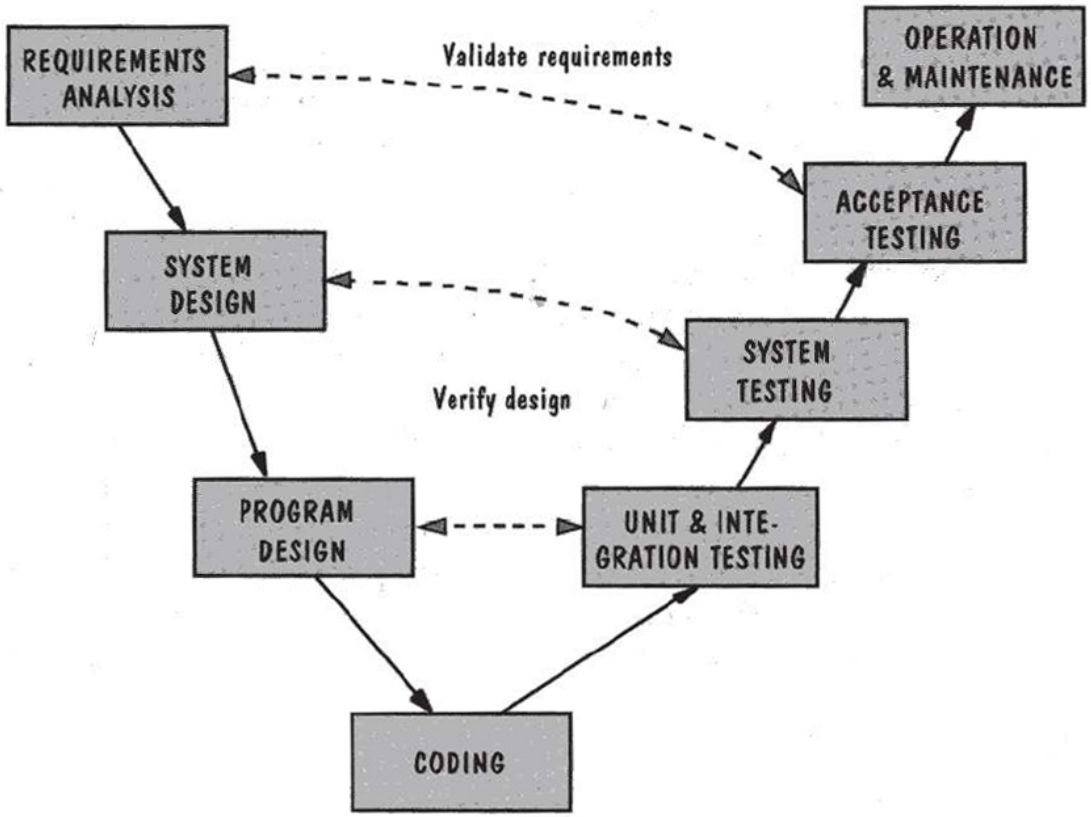
Reality is more like this:



[1]Figures taken from: Shari Lawrence Pleeger: Software Engineering, 2. Edition, Prentice-Hall 2001.

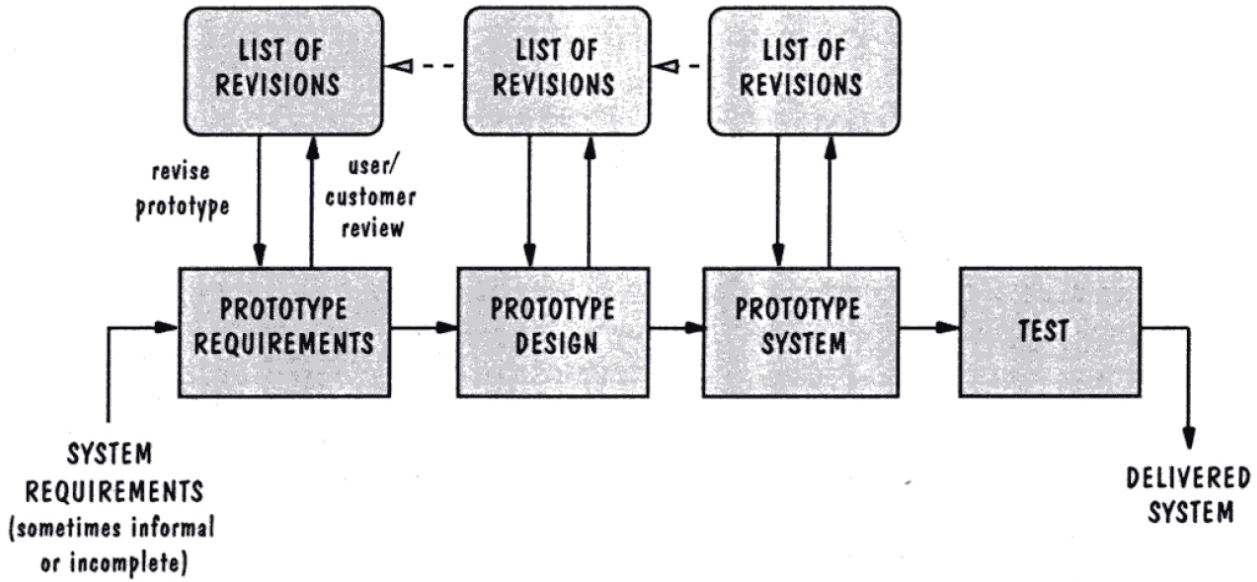
Alternatives to the Waterfall Model

The V-Model



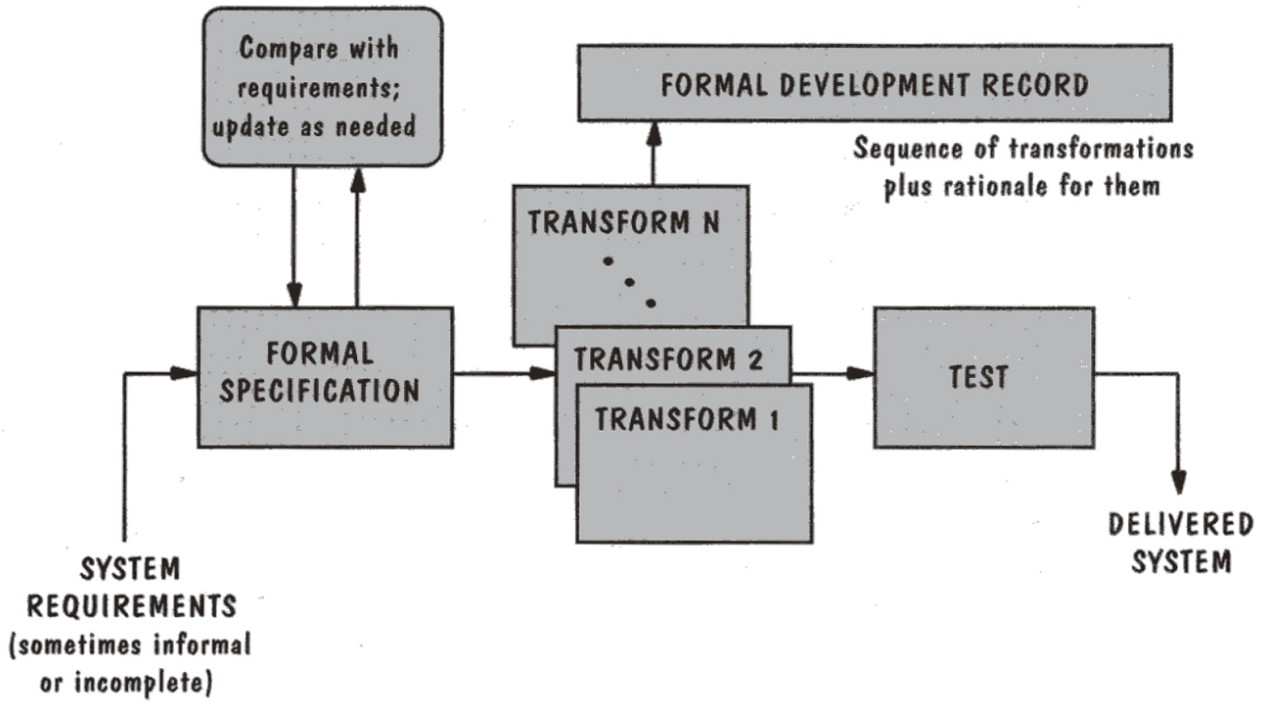
Explicitly relates validation and development phases.

The Prototype Model



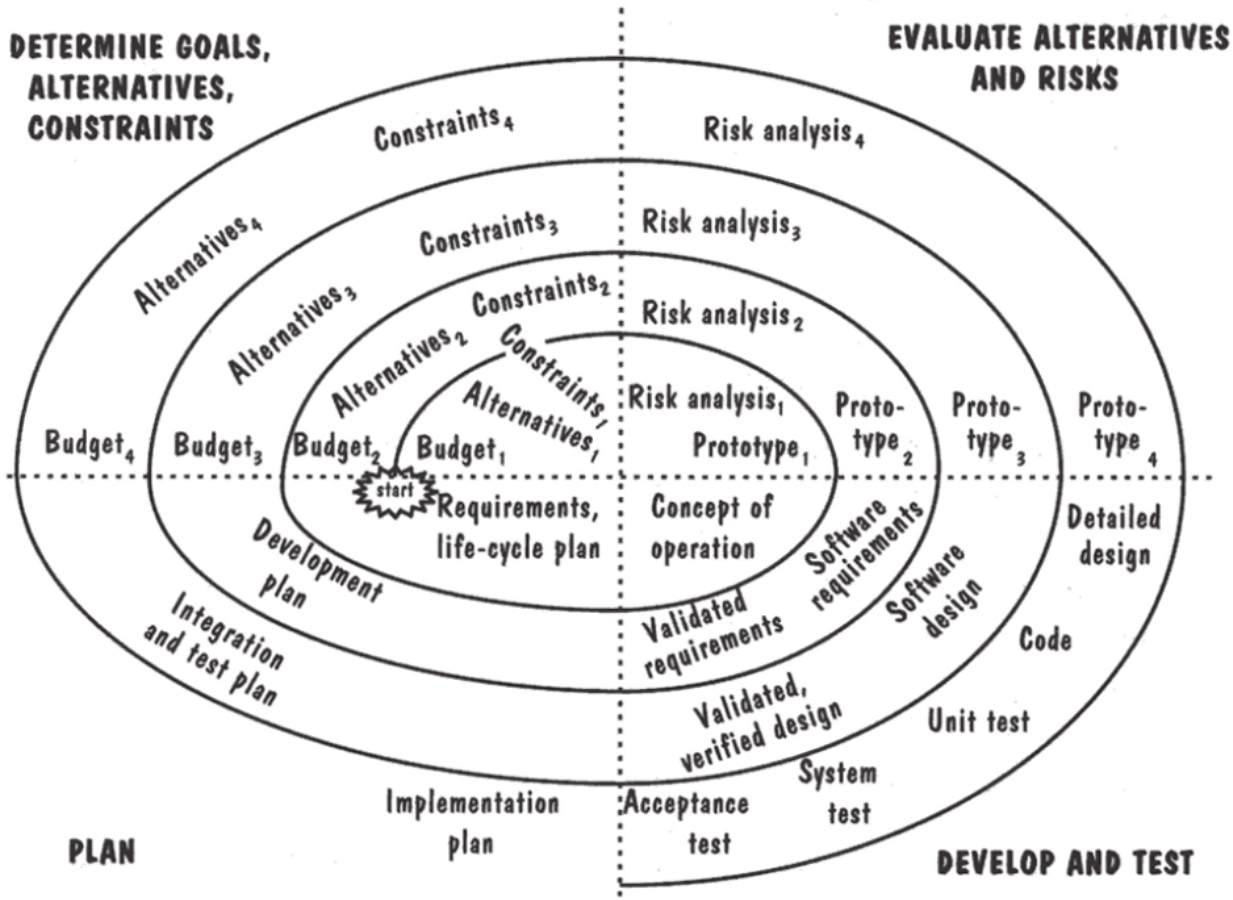
Good for systems, whose requirements are not clear from the beginning. Can lead to poorly structured systems.

The Transformation Model



In principal, a good idea, the practical implementation of the transformation however is a problem.

The Spiral Model



Incremental development, risk-driven.

Phases of software engineering processes

- Analysis
  Goal: understand the problem
- Design
  Goal: obtain structure of software to be built
- Implementation
  Goal: obtain executable software solving the problem
- Testing
  Goal: find defects in implementation

Conclusion

All software process models essentially contain the same **phases**, which however are organized differently.

- **Analysis/Requirements Definition**: describes, what the system should do. Cooperation among developers and users.
- **Design**: defines the **architecture of the system**. Represent the system functions in such a way that they can be transformed into programs.
- **Implementation**: realizes the design through program units.
- **Validation**: consists usually tests and reviews, proofs also possible.
- **Service and Maintenance**: removal of errors, improvement of the functionality.

$\Rightarrow$ Programming represents only one of many phases/actions!

ADIT Development Process

Overview of analysis steps

A1. Problem elicitation and description

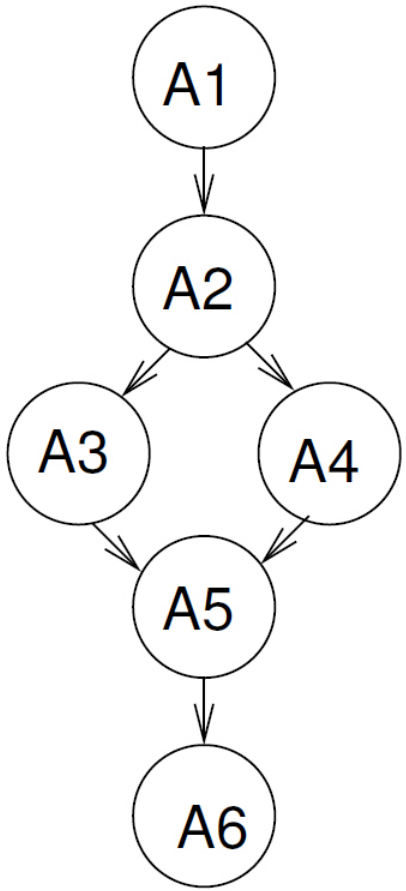
A2. Problem decomposition and patterns

A3. Abstract software specification

A4. Technical software specification

A5. Operations and data specification

A6. Software life-cycle



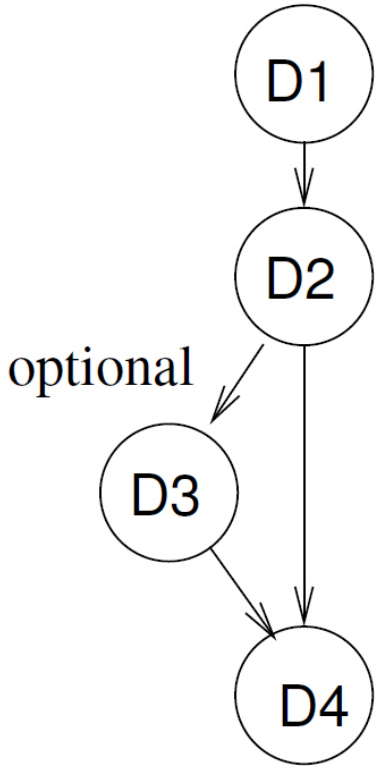
Overview of design steps

D1. Software architecture

D2. Inter-component-interaction

D3. Intra-component-interaction

D4. Complete component or class behaviour



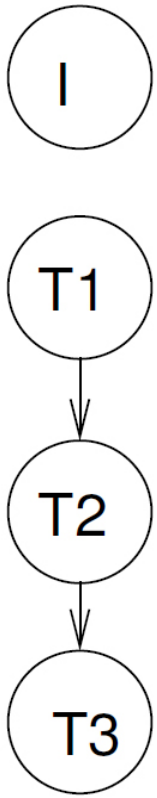
Overview of implementation and testing steps

I. Implementation and unit test

T1. Component test

T2. System test

T3. Acceptance test



Patterns and Components

Why patterns and components?

- Both are promising new developments.
- Both are based on reuse:
  - Patterns allow re-use of **software development knowledge**.
  - Components allow re-use of **pre-fabricated software**.
- Both can be used in combination.
- Both fit well with **model-based development**, where sequences of models are set up that highlight different aspects of the software and that area of a decreasing level of abstraction.

Patterns: basic ideas

- **Templates** for documents set up during software development
- Serve to represent and re-use software development knowledge
- Represent **essence**, abstract from details
- Are used by **instantiation**
- Are available for (almost) all phases of software development

Components: basic ideas

- Assemble software from pre-fabricated parts
- Re-compilation not necessary
- Source code may be inaccessible
- Important: interface descriptions and component models
- Interoperability is an issue

Basic Terminology

Terminology I

Goal

Construction of a system with specified characteristics

Example: An elevator should enable persons in a building to get from one floor to another.

Components of the system:

- environment: part of "real world" relevant for the problem
  Example: floors, persons, cage, doors, engine, buttons, sensors, ...
- machine: software (running on some hardware)

Properties of the environment are mostly fixed. We have to build the machine, so that it realizes the desired properties of the system.

Terminology II

State and behaviour of the environment can be described by phenomena. Examples:

- **Elevator**
  Person *presses button*, expects, that the *elevator arrives*.
- **Bank**
  Client gives *withdrawal instruction*, expects, a *withdrawal*.

Machine can interact with the environment by

- observing certain phenomena (input)
- causing certain phenomena (output)

Terminology III

Phenomena are

- events, actions or operations that occur in the environment
- important for expressing **statements**
- can be observed or controlled by the environment or the machine, respectively

Examples:

- waiting in front of the elevator
- pressing the button inside the elevator
- elevator door closes

Terminology IV

Categories of phenomena

1. Controlled by the environment, not observable by the machine
   Example: *waiting in front of elevator*
2. Controlled by the environment, observable by the machine
   Example: *pressing the button inside the elevator*
3. Controlled by the machine, observable by the environment
   Example: *elevator door closes*

The category "controlled by the machine, not observable by the environment" is not considered, since internal phenomena of the machine do not belong to the requirements.

Requirements and Domain Knowledge

Role of requirements and domain knowledge

- Known:
  (1) Characteristics of the environment (**domain knowledge**)
  (2) Desired characteristics of the system (**requirements**)
- Machine must close the "gap" between (1) and (2)
- Searched: **specification** for the machine
  "How should the machine act, so that the system fulfills the requirements?"

Note:

Requirements describe the *environment*, the way it should be, after the machine is integrated.

Type of statements I

Note: Statements are characterized by being *true* or *false*.

Optative Statements describe the environment, in the way we would expect it after the machine is integrated.

Example: After the button was pressed, the elevator stops at the according floor.

Requirements are thus optative statements.

Indicative Statements describe the environment. Other notation: **domain knowledge**.

We distinguish two kinds of domain knowledge: *facts* and *assumptions*.

Type of statements II

Facts describe fixed properties of the environment irrespective of how the machine is built.

Example: a door can not be open and closed at the same time.

Assumptions describe conditions that are needed, so that the requirements are accomplishable.

Example: *When the elevator reaches my floor, I enter*.

This assumption is needed for fulfilling the requirement that the elevator carries all waiting persons to their destination.

Requirements vs. Specifications

Specifications

- are descriptions that are sufficient for building the machine
- are **implementable** requirements

Correctness condition:

If the machine fulfils the specification, the system fulfils the requirements.

Requirements are **NOT** implementable, if they

- constrain phenomena that are controlled by the environment

  Example: *The elevator is not to be overloaded*
- refer to phenomena that are not observable by the machine

  Example: *The elevator should go to the floor where people are waiting.*
- make constraints for the future

  Example: *As soon as a user has dialed the last digit, he receives the dial tone, the busy signal, or the announcement "number not assigned".*

Deriving specifications from requirements I

- Searched: specification of the machine
- Known: facts F, assumption A and requirements R
- Question: how do you derive the specification S, such that $D \wedge S \Rightarrow R$ (**correctness of the specification**) where $D \equiv F \wedge A$
- Answer: use domain knowledge to
  - replace phenomena that are not observable by the machine by related ones that are observable
  - replace phenomena that are not controlled by the machine by related ones that are controlled by it
  - replace requirements that refer to the future by **invariants**

Examples

Requirement:
The number of persons who enter the zoo is counted.
Domain knowledge:
Persons who enter the zoo push a turnstyle.
Specification:
The number of turnstyle pushes are counted.

Requirement:
No more than n persons may enter the zoo per day.
Domain knowledge:
The turnstyle can be locked by the machine.
Specification:
The turnstyle should be locked after n pushes are counted.

Correctness criteria

Consider requirements R, facts F, assumptions A, specification S.

1. Each element of R is considered acceptable by the client, and R contains all customer preferences concerning the software development project.
2. Each element of F was checked for correctness.
3. Each element of A was checked for plausibility.
4. All elements of S are implementable.
5. $D \wedge S \Rightarrow R$ is demonstrated.
6. It is proven that F, A, and S are consistent.

When these criteria are fulfilled, the the construction of a machine that fulfills S can be started.

Contoh: Penyewaan Villa

Requirements and domain knowledge

Deskripsi

Sebuah sistem penyewaan rumah/villa untuk liburan memiliki fitur yang memungkinkan tamu untuk memilih dan memesan paket liburan yang ditawarkan. Setelah memesan, tamu selanjutnya akan menerima tagihan dan pemesanan selanjutnya akan disimpan. Tamu memiliki waktu 14 hari untuk membayar tagihan melalui transfer bank. Jika melewati batas waktu tersebut maka secara otomatis paket liburan yang telah dipesan akan ditawarkan kembali ke tamu yang lainnya. Anggota staf berkewajiban untuk mencatat semua pembayaran yang masuk dan juga mengecek kondisi rumah setelah ditinggalkan oleh tamu. Jika kondisi rumah setelah ditinggalkan tamu buruk/rusak, maka tamu akan dikenakan tagihan tambahan. Kewajiban lainnya dari staf adalah memasukkan penawaran untuk paket liburan yang baru.

Requirements untuk penyewaan villa I

(R01) Staf meberikan penawaran paket liburan.

(R02) Tamu dapat memilih paket yang ditawarkan.

(R03) Tamu dapat memesan paket yang ditawarkan dan pemesanan akan disimpan.

(R04) Setelah tamu memesan paket, tagihan akan dikirim.

(R05) Jika pembayaran tidak dilakukan dalam 14 hari, maka paket akan ditawarkan ke tamu yang lain.

Requirements untuk penyewaan villa II

(R06) Staf mencatat semua pembayaran yang diterima.

(R07) Staf mengecek kondisi rumah setelah tamu pergi.

(R08) Jika kondisi rumah yang ditinggalkan buruk/rusak, tamu akan dikenakan tagihan tambahan.

(R09) Staf dapat melihat tawaran yang telah dipesan/-booking oleh tamu.

Domain knowledge untuk penyewaan villa I

(F01) Villa hanya boleh digunakan oleh satu tamu (beserta keluarga/temannya) pada waktu yang bersamaan.

(F02) Alamat villa unik.

(F03) Villa yang ditawarkan memiliki alamat, kapasitas, waktu sewa, dan harga sewa per hari.

(A01) Tamu akan membayar tagihan dalam waktu 14 hari.

(A02) Tamu yang telah membayar artinya jadi menyewa villa dan berlibur.

(A03) Pengecekan kondisi villa oleh staf dilakukan secara fair.

Domain knowledge untuk penyewaan villa II

(A04) Staf selalu up-to-date dengan pembayaran yang masuk.

(A05) Sudah ada kontrak antara pemilik villa dan pemilik sistem penyewaan.

(A06) Staf hanya dapat memasukkan paket yang sama sekali.

(A07) Tagihan yang dikirim akan sampai kepada pemesan yang bersangkutan.

(A08) Tamu akan meninggalkan villa setelah masa sewa berakhir.

Summary of the terminology

You should have learned the following notations:

- machine
- phenomenon
- environment
- action/event/operation
- indicative statement
- requirement
- optative statement
- specification
- assumption
- fact

What have we learned? I

- The task in system and software development is to build a *machine*. This machine is integrated in an *environment*, in order to improve the behaviour of the environment.
- (Functional) requirements describe the desired behaviour of the environment after the machine has been integrated. They do *not* describe the machine.
- Requirements are expressed as *optative statements*.
- *Indicative statements* describe the environment (by facts and assumptions).
- *Facts* are fixed properties of the environment; they are always true, no matter how we build the machine.
- *Assumptions* are sometimes necessary for realizing the requirements.

What have we learned? II

- Machine and environment can interact by means of *shared phenomena*. These phenomena are either controlled by the environment or by the machine.
- *Specifications* are implementable requirements.
- In order to convert requirements into specifications, facts and assumptions (indicative statements) are used.

Problem Frames

Problem Frames I

- Are **patterns** that characterize frequently occuring problems
- Simple subproblems resulting from problem decomposition should fit to a problem frame
- Are represented with **frame diagrams**
- Concrete problems are "fitted to problem frames"
- Five basic problem frames Jackson (2001)
  - required behaviour
  - **commanded behaviour**
  - information display
  - **simple workpieces**
  - transformation

Problem Frames II

- Some other frames are relevant
- Metamodel for problem frames
- Can be represented using UML stereotypes
- Tool support available
- Exist also for non-functional requirement

Domain Types I

Problem frames structure the environment by means of **domains**. These can have different types:

Causal Properties include predictable causal relationships among its causal phenomena. Often, causal domains are mechanical or electrical equipment. They are indicated with a "C" in frame diagrams.

Lexical Physical representation of data – that is, of symbolic phenomena. It combines causal and symbolic phenomena in special way. The causal properties allow the data to be written and read. Lexical domains are indicated by "X".

Domain Types II

Biddable Usually consists of people. The most important characteristic of biddable domain that it's physical but lacks positive predictable internal causality. That is, in most situations it's impossible to compel a person to initiate an event: the most that can be done is to issue instructions to be followed. Biddable domains are indicated by "B".

Others E.g., display domains

Problem frame: commanded behaviour I

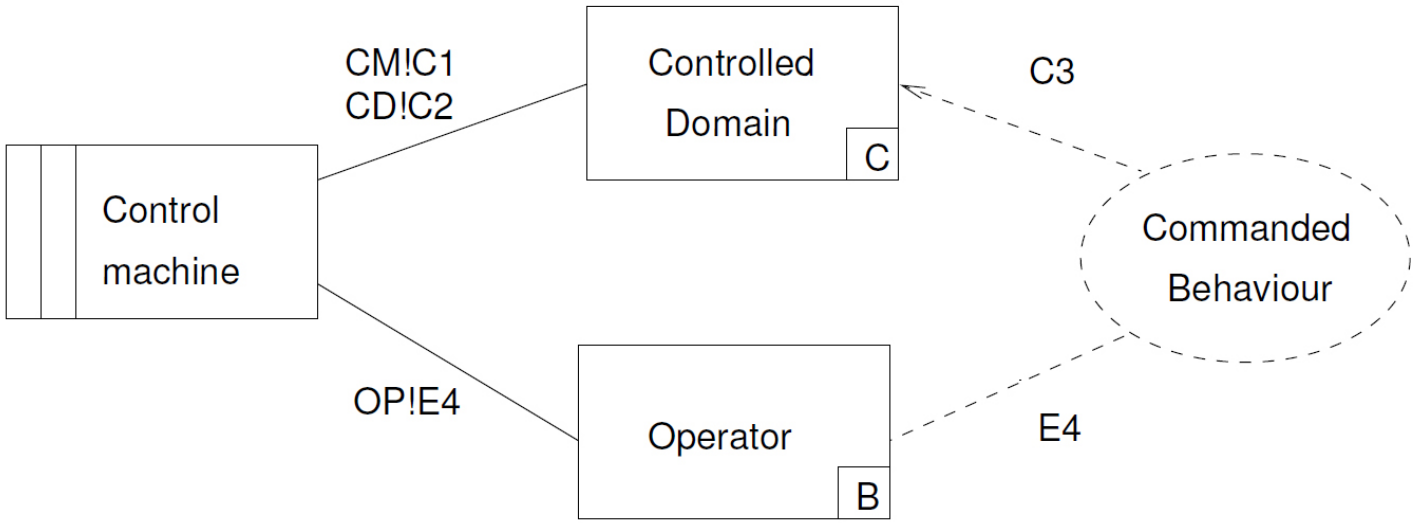
Informal description

There is some part of the physical world whose behaviour is to be controlled with commands issued by an operator. The problem is to build a machine that will accept the operator's commands and impose the control accordingly.

An operator can issue commands that influence the behaviour of the controlled domain. Usually the operator knows that he/she is an operator.

Problem frame: commanded behaviour II

Frame diagram



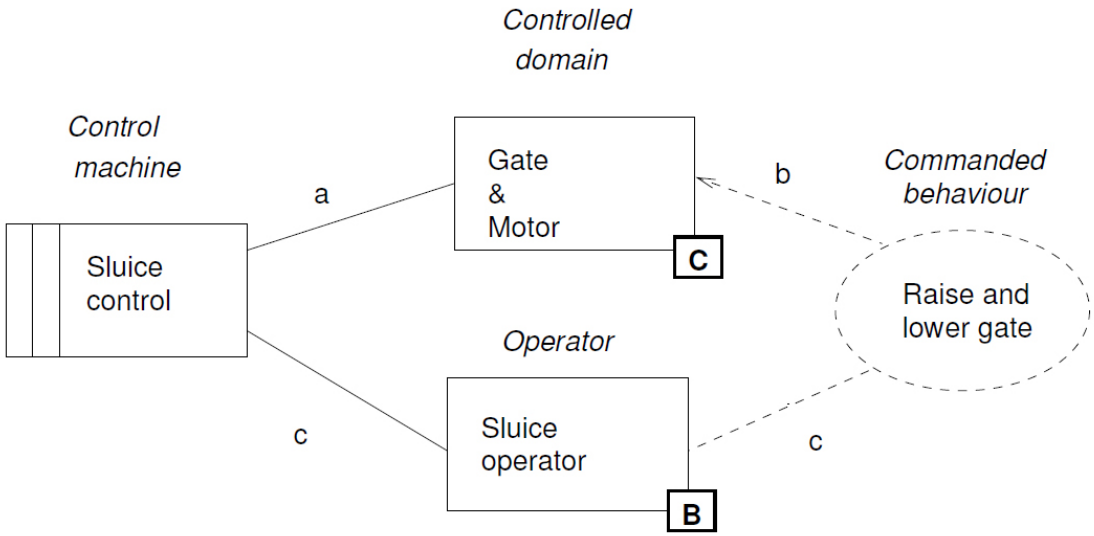
B: A **biddable** domain cannot be constrained by a machine
E4: operator commands
Determine how and when the machine should – or should not – cause C1-phenomena in response to E4 commands.

Example: occasional sluice gate control I

A small sluice is used in a simple irrigation system. The sluice gate can rise and fall. **A computer system is needed to raise and lower the sluice gate in response to the commands of the operator**.
The gate is open and closed by rotating vertical screws. The screws are driven by a small motor, which can be controlled by clockwise, anticlockwise, on and off pulses. There are sensors at the top and bottom of the gate travel; at the top it's fully open, at the bottom it's fully shut. The connection to the computer consists of four pulse lines for the gate sensor, **a status line for each class of operator command**.

Example: occasional sluice gate control II

Fitting the occasional sluice gate control to the commanded behaviour frame



a: SC!{Clockw, Anti, On, Off} [C1]
GM!{Top, Bottom} [C2]
b: GM!{Open, Shut, Rising, Falling} [C3]
c: SO!{Raise, Lower, Stop} [E4]

Fitting problems to problem frames is achieved by **instantiating** the variables contained in the frame diagram.

Problem frame: simple workpieces I

Informal description

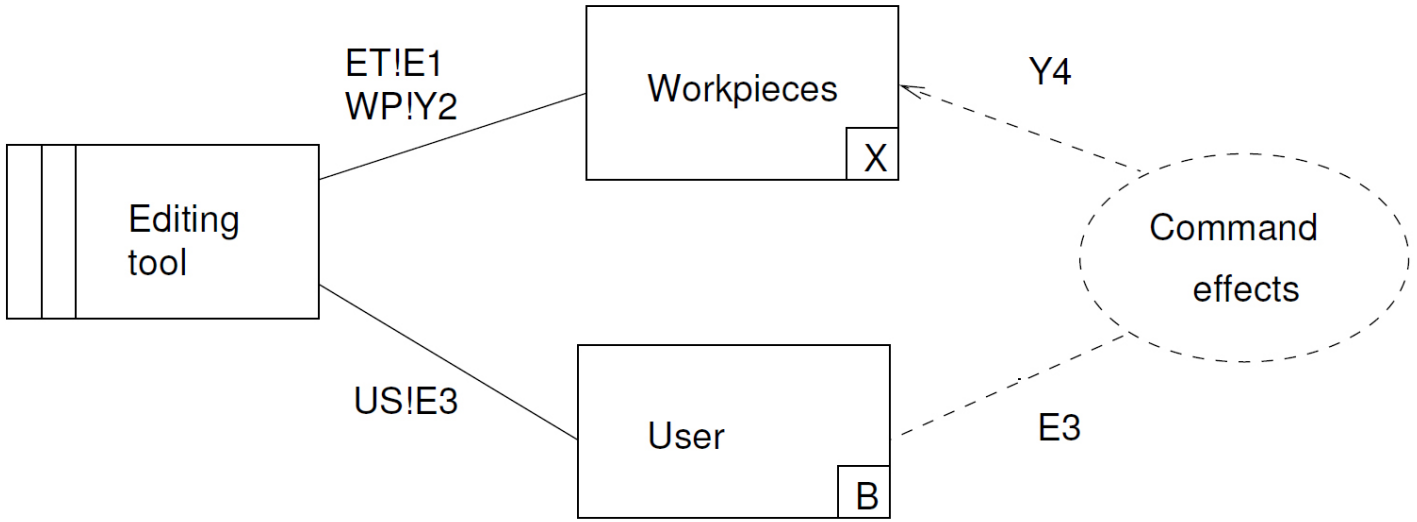
A tool is needed to allow a user to create and edit a certain class of computer-processable text or graphic objects, or similar structures, so that they can be subsequently copied, printed, analysed or used in other ways. The problem is to build a machine that can act as this tool.

Workpieces:
Normally a piece of material worked on by a machine, e.g. wood blocks, metal casting, etc.

Here: things that are worked on by a computer.

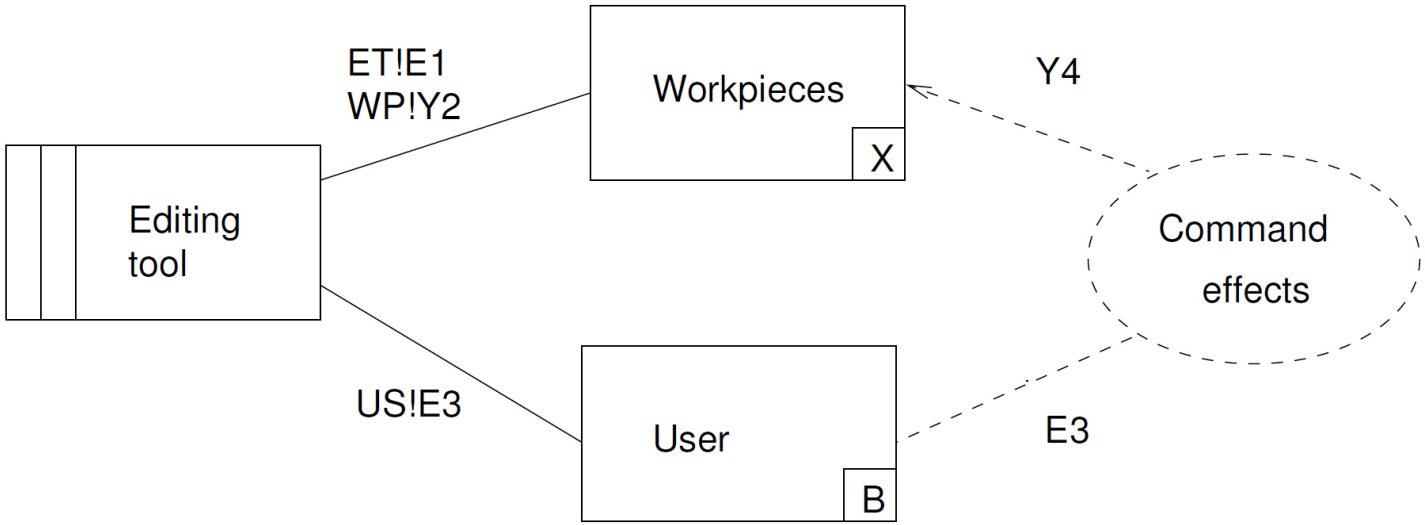
Problem frame: simple workpieces II

Frame diagram



X: **lexical** domain, i.e. data type
Phenomena E1: *Operations* on workpieces
Phenomena Y2: allow the machine to examine the current state and values of the workpiece.

Problem frame: simple workpieces III

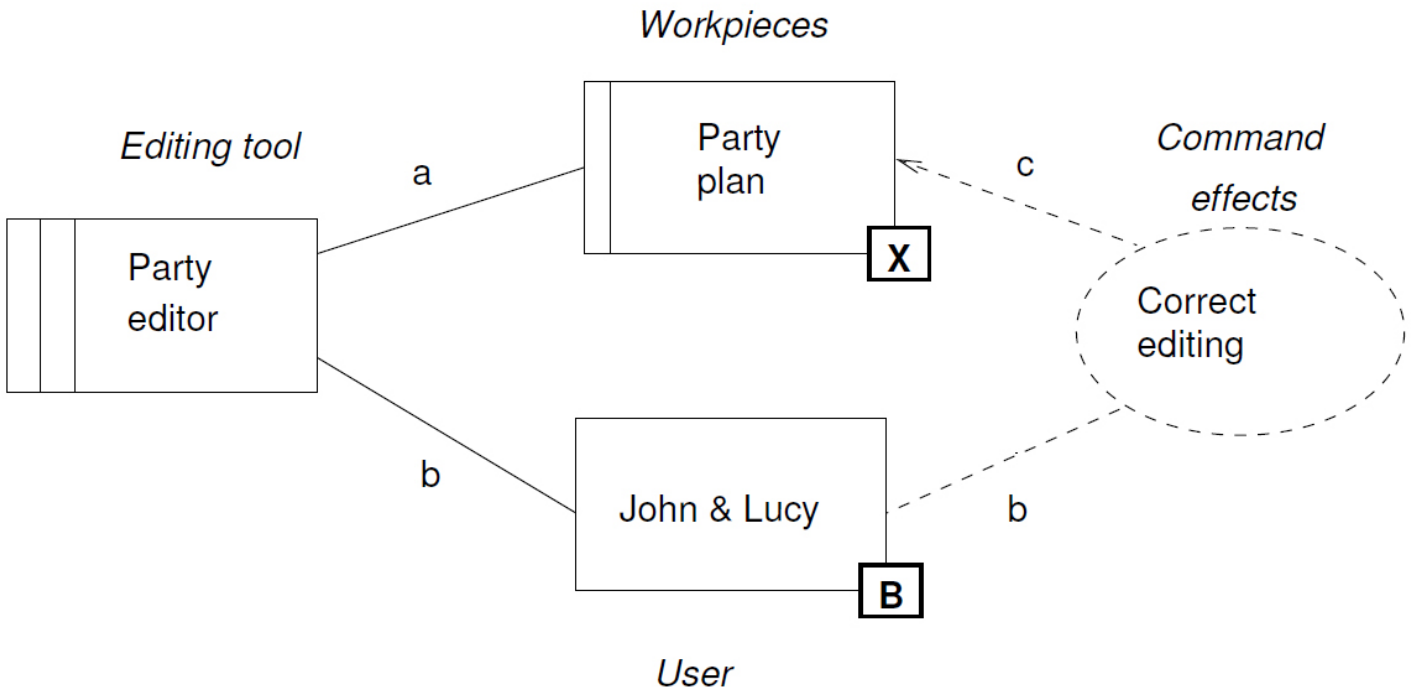


Phenomena E3: user commands
Requirements: describe effects, commands E3 should have on phenomena Y4.
Y4 often differs from Y2: Phenomena Y4 can have some meaning to the user that is insignificant for editing.
Simple workpieces does **not** only deal with printing, representing or further processing the workpieces but also for problems identification.

Example: party plan I

Lucy and John need a system to keep track of the many parties they give and the many guests they invite. They want a simple editor to maintain the information, which they call their *party plan*. Essentially the party plan is just a list of parties, a list of guests, and a note of who is invited to each party. The editor will accept command-line text input.

Example: party plan II



a: PE!{PlanOperations} [E1]
PP!{PlanStates} [Y2]
b: JL!{Commands} [E3]
c:PP!{PlanEffects} [Y4]

Remark

The structure of the problem frames simple workpieces and commanded behaviour are **very similar!**
Differences:

- The Workpieces domain is lexical and formal, but the controlled domain is causal and informal.
- In the controlled domain, approximations must be taken into account, which is unnecessary for the Workpieces domain.
- The requirements in the simple workpieces frame relate only to the user's commands, while in the commanded behaviour frame other requirements can occur, e.g. that the sluice gate is not driven beyond the limits of its vertical travel.

Working with problem frames (simplified process)

1. Describe problem
   - state requirements
   - state domain knowledge
   - identify relevant domains and their shared phenomena
2. Divide problem into subproblems
   - identify sets of related requirements (e.g. by use-case decomposition or projection)
   - subproblems should be *simple*
   - fit subproblems to problem frames
3. Derive machine specification for each subproblem
4. Set up software architecture for each subproblem machine (according to patterns defined for each problem frame)
5. Merge software architecture (according to rules)
6. Implement, test and integrate components of that architecture

"Completeness" of problem frames

- Besides the basic problem frames, there are other useful problem frames, which can be found systematically.
- The problem-frame-approach can also be used to analyze non-functional requirements.
  Example: usability, security.

Finding further problem frames

"Complete" enumerations of possible problem frames, according to domain types and number of involved domains.

Note: introducing a new domain type would yield new possible frame candidates.

The legend for the following tables:

n/a not applicable. The problem frame is in conflict with an integrity condition.

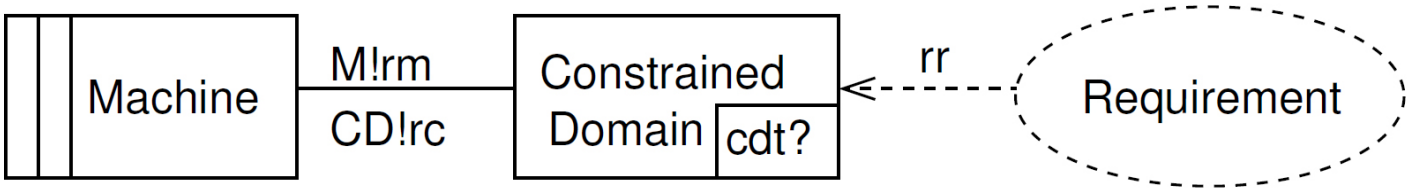
o Not many applications using this problem frame was found.

+ A useful frame; it is usually used in combination with other frames and is applicable to many realistic applications.

++ A very useful frame; this frame can be used for many realistic applications.

One (constrained) domain I

Frame diagrams containing only one constrained problem domain:



One (constrained) domain II

Problem frames with one problem domain:

No.	Constrained domain type (cdt?)	Comment	Author	Relevance
PF 1.1	B	*Biddable cannot be constrained*	-	n/a
PF 1.2	C	**required behaviour**	Jackson (2001)	++
PF 1.3	D	generated information	Côté et al. (2008)	o
PF 1.4	**X**	**simple transformation**	**Côté et al. (2008)**	+

One (constrained) domain III

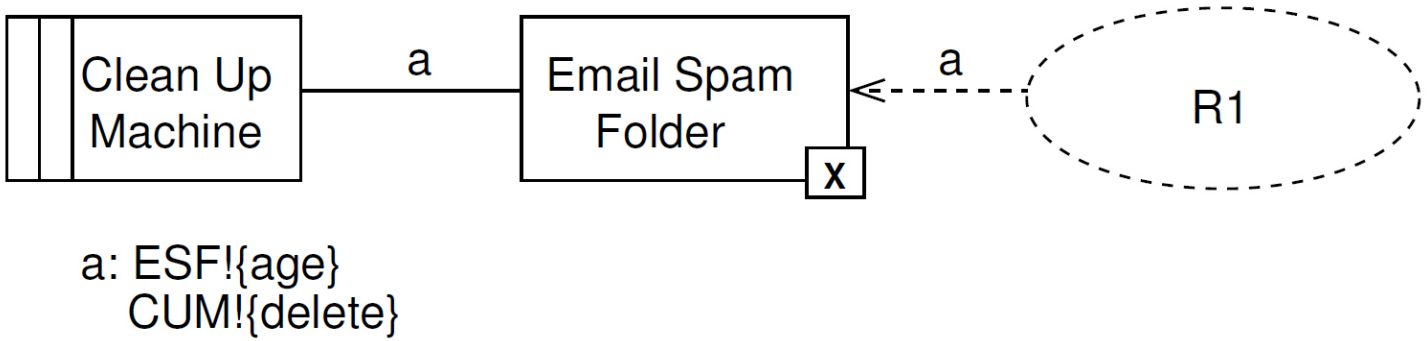
Simple Transformation - Informal Descripton

A file is cyclically or automatically changed. It is similar to the transformation frame and a special case of a required behaviour frame. An example is a counter or any automatic procedure changing a lexical domain.

Frame Diagram

One (constrained) domain IV

Instantiated Frame Diagram - Example: Automatic delete

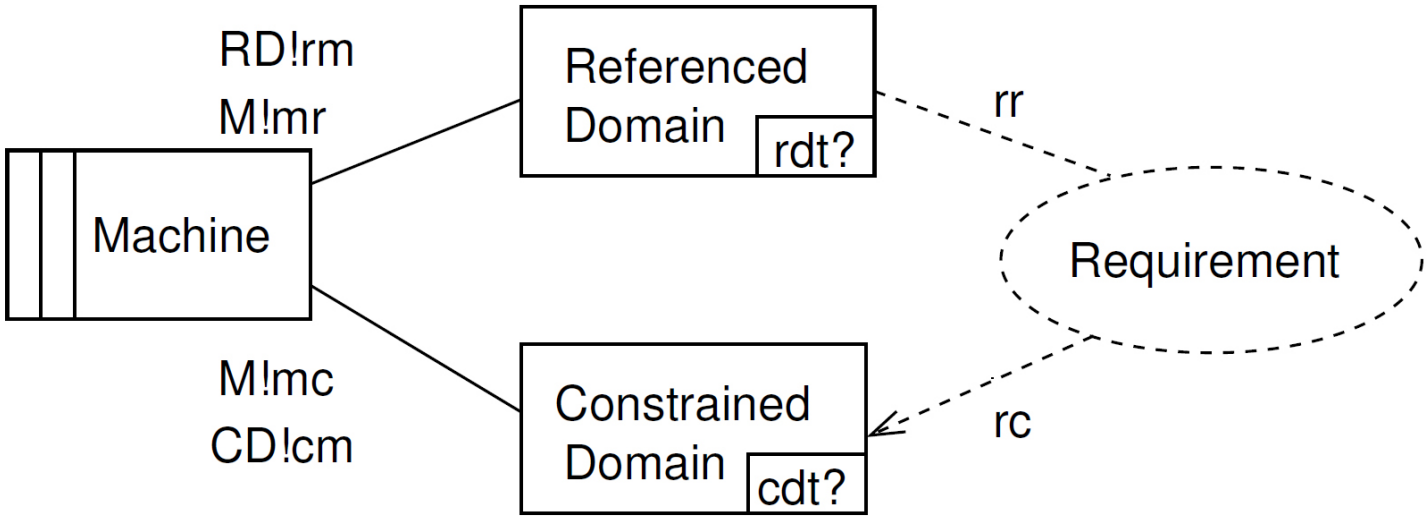


R1: Delete all messages from the email spam folder that are older than 14 days.

The lexical domain email spam folder is constrained by this requirement.

Two domains, one constrained I

Frame diagrams with two problem domains, one of which is constrained:



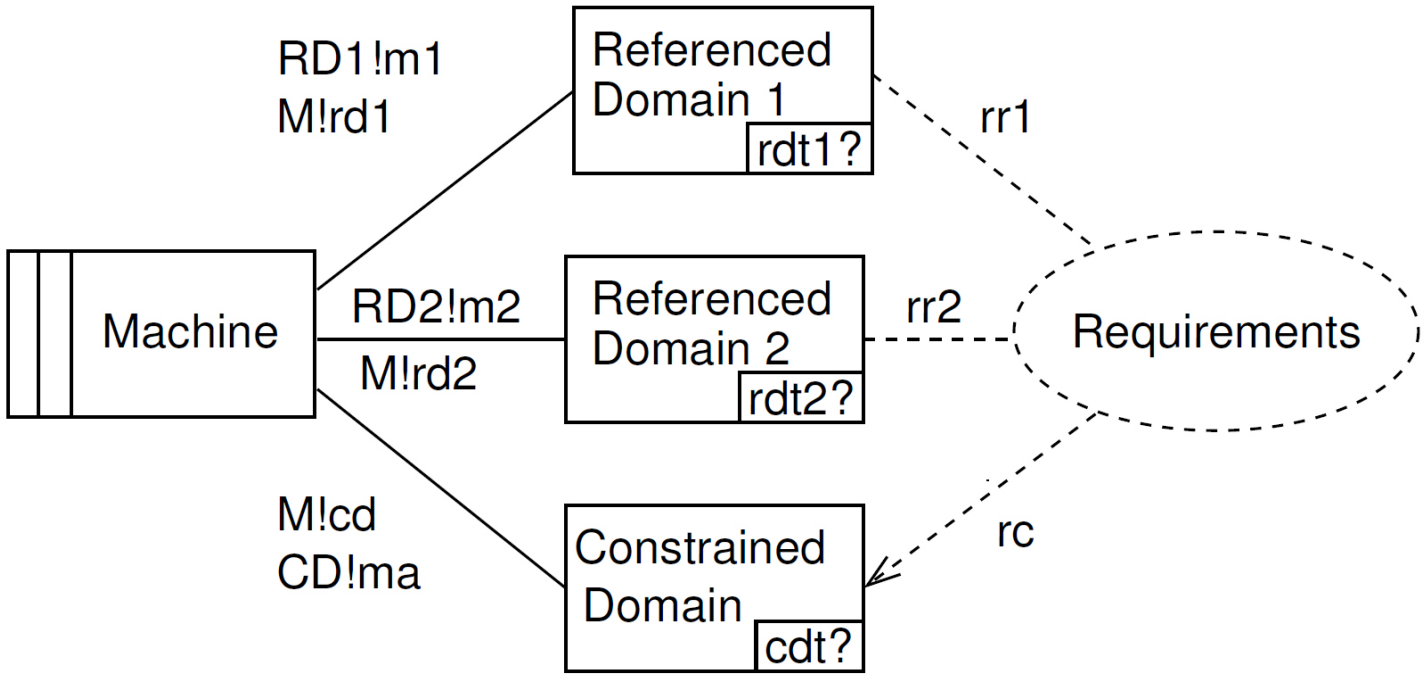
Two domains, one constrained II

Problem frames with two problem domains:

No.	Ref. dom. type (rdt1?)	Constr. dom. type (cdt?)	Comment	Author	Relevance
PF 2.1	X	X	**transformation**	Jackson (2001)	++
PF 2.2	**X**	**C**	**data-based control**	**Côté et al. (2008)**	**+**
PF 2.3	X	D	model display	Jackson (2001)	+
PF 2.4	C	X	model building	Jackson (2001)	+
PF 2.5	C	C	= required behaviour (merge)	Jackson (2001)	see PF 1.2
PF 2.6	C	D	**information display**	Jackson (2001)	++
PF 2.7	B	X	**simple workpieces**	Jackson (2001)	++
PF 2.8	B	C	**commanded behaviour**	Jackson (2001)	++
PF 2.9	B	D	commanded display	Côté et al. (2008)	+
PF 2.10	*	B	*Biddable cannot be constrained*	-	n/a
PF 2.11	D	*	*Display must be constrained*	-	n/a

Three domains, one constrained I

Frame diagrams with three problem domains, one of which is constrained:



Three domains, one constrained II

Problem frames with three problem domains (one constrained, part 1):

No.	Ref. dom. 1 type (rdt1?)	Ref. dom. 2 type (rdt2?)	Constr. dom. type (cdt?)	Comment	Author	Relevance
PF 3.1	X	X	X	= transformation (merge)	Jackson (2001)	PF 2.1
PF 3.2	C	X	X	= model building (merge)	Jackson (2001)	PF 2.4
PF 3.3	**B**	**X**	**X**	**commanded transformation**	**Wentzlaff and Specker (2006)**	++
				= simple workpieces, *if referenced domains can be merged*	Jackson (2001)	PF 2.7
PF 3.4	D	*	*	*Display must be constrained*	-	n/a
PF 3.5	X	C	X	(interchange) PF 3.2	-	PF 3.2
PF 3.6	C	C	X	= model building (merge)	Jackson (2001)	PF 2.4
PF 3.7	B	C	X	commanded model building = simple workpieces + model building	Côté et al. (2008)	+
PF 3.8	X	B	X	(interchange) PF 3.3	-	PF 3.3
PF 3.9	C	B	X	(interchange) PF 3.7	-	PF 3.7
PF 3.10	B	B	X	multi-user simple workpieces = simple workpieces + simple workpieces	Côté et al. (2008)	+
PF 3.11	*	D	*	*Display must be constrained*	-	n/a

Three domains, one constrained III

Problem frames with three problem domains (one constrained, part 2):

No.	Ref. dom. 1 type (rdt1?)	Ref. dom. 2 type (rdt2?)	Constr. dom. type (cdt?)	Comment	Author	Relevance
PF 3.12	X	X	C	= data-based control (merge)	-	PF 2.2
PF 3.13	C	X	C	(interchange) PF 3.15	-	PF 3.15
PF 3.14	B	X	C	commanded data-based control = commanded behaviour + data-based control	Côté et al. (2008)	++
PF 3.15	X	C	C	= data-based control (merge)	-	PF 2.2
PF 3.16	C	C	C	= required behaviour (merge)	Jackson (2001)	PF 1.2
PF 3.17	B	C	C	= commanded behaviour (merge)	Jackson (2001)	PF 2.8
PF 3.18	X	B	C	(interchange) PF 3.14	-	PF 3.14
PF 3.19	C	B	C	(interchange) PF 3.17	-	PF 3.17
PF 3.20	B	B	C	multi-user commanded behaviour = commanded behaviour + commanded behaviour	Côté et al. (2008)	+
PF 3.21	*	*	B	*Biddadble cannot be constrained*	-	n/a
PF 3.22	X	X	D	= model display (merge)	-	PF 2.3
PF 3.23	C	X	D	= information display (PF 2.6) + model display (PF 2.3)	Côté et al. (2008)	o

Three domains, one constrained IV

Problem frames with three problem domains (one constrained, part 3):

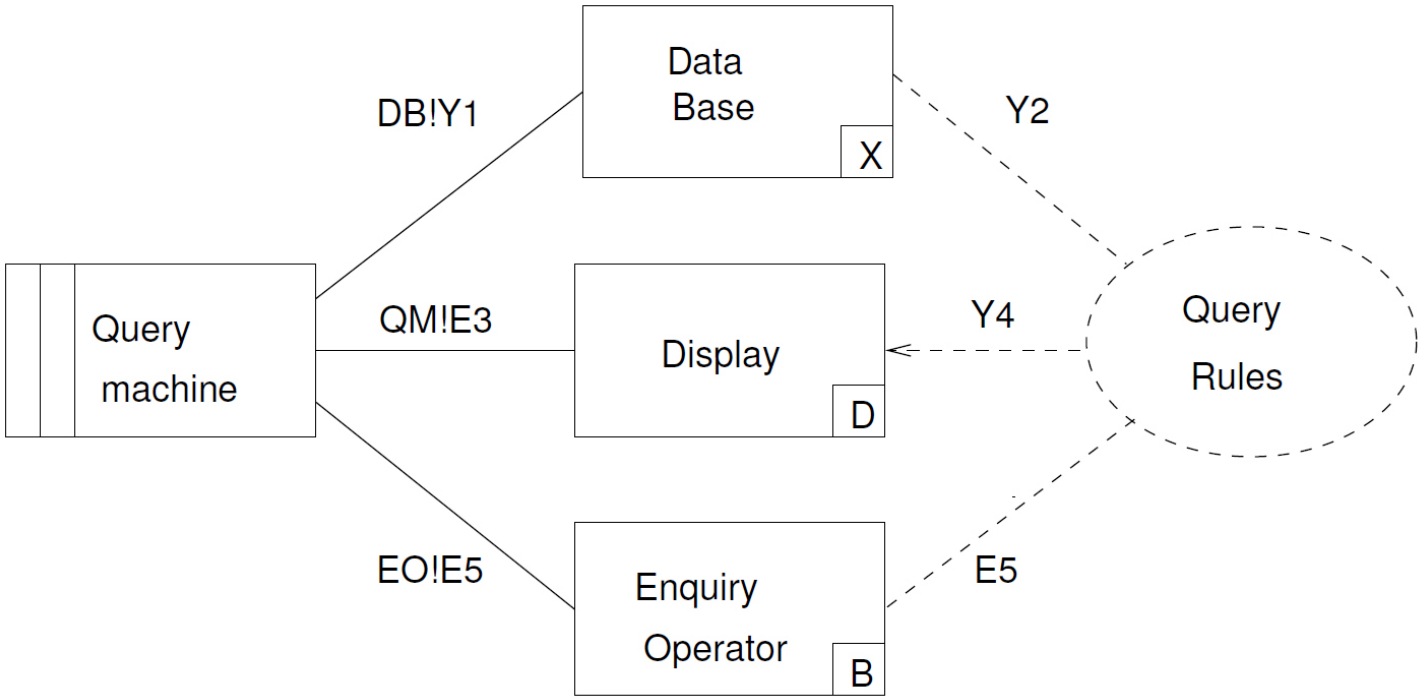
No.	Ref. dom. 1 type (rdt1?)	Ref. dom. 2 type (rdt2?)	Constr. dom. type (cdt?)	Comment	Author	Relevance
PF 3.24	B	X	D	query = commanded display + model display	Choppy and Heisel (2004), Wentzlaff and Specker (2006)	+
PF 3.25	X	C	D	(interchange) PF 3.23	-	PF 3.23
PF 3.26	C	C	D	= information display (merge)	-	PF 2.6
PF 3.27	B	C	D	commanded information = information display + commanded behaviour	Jackson (2001)	++
PF 3.28	X	B	D	(interchange) PF 3.24	-	PF 3.24
PF 3.29	C	B	D	(interchange) PF 3.27	-	PF 3.27
PF 3.30	B	B	D	multi-user commanded display = commanded display + commanded display	Côté et al. (2008)	+

Three domains, one constrained V

Query - Informal Description

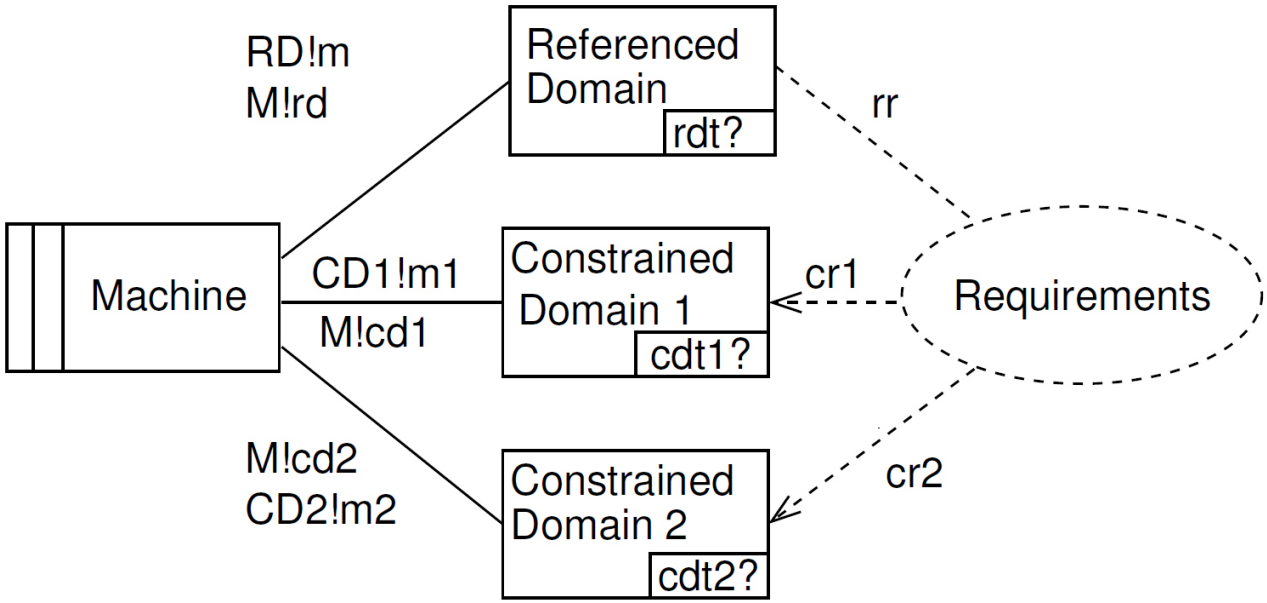
An operator queries information from database. This information is displayed for the operator.

Frame Diagram



Three domains, two constrained I

Frame diagrams with three problem domains, two of which are constrained



Three domains, two constrained II

Problem frames with three problem domains (two constrained, part 1):

No.	Ref. dom. type (rdt?)	Constr. dom. 1 type (cdt1?)	Constr. dom. 2 type (cdt2?)	Comment	Author	Relevance
PF 4.1	X	X	X	= transformation (merge)	Jackson (2001)	PF 2.1
PF 4.2	C	X	X	= model building (merge)	Jackson (2001)	PF 2.4
PF 4.3	B	X	X	= simple workpieces (merge)	Jackson (2001)	PF 2.7
PF 4.4	D	*	*	*Display must be constrained*	-	n/a
PF 4.5	X	C	X	(interchange) PF 4.11	-	PF 4.11
PF 4.6	C	C	X	(interchange) PF 4.12	-	PF 4.12
PF 4.7	B	C	X	(interchange) PF 4.13	-	PF 4.13
PF 4.8	X	D	X	model display (PF 2.3) + transformation (PF 2.1)	Côté et al. (2008)	o
PF 4.9	C	D	X	information display (PF 2.6) + model building (PF 2.4)	Côté et al. (2008)	o
PF 4.10	**B**	**D**	**X**	**update = simple workpieces + model display**	**Choppy and Heisel (2004)**	++
PF 4.11	X	X	C	data-based control (PF 2.2) + transformation (PF 2.1)	Côté et al. (2008)	o
PF 4.12	C	X	C	model building (PF 2.4) + required behaviour (merge) (PF 1.2)	Côté et al. (2008)	o

Three domains, two constrained III

Problem frames with three problem domains (two constrained, part 2):

No.	Ref. dom. type (rdt?)	Constr. dom. 1 type (cdt1?)	Constr. dom. 2 type (cdt2?)	Comment	Author	Relevance
PF 4.13	B	X	C	simple workpieces (PF 2.7) + commanded behaviour (PF 2.8)	Côté et al. (2008)	o
PF 4.14	X	C	C	= data-based control (merge)	Côté et al. (2008)	PF 2.2
PF 4.15	C	C	C	= required behaviour (merge)	Jackson (2001)	PF 1.2
PF 4.16	B	C	C	= commanded behaviour (merge)	Jackson (2001)	PF 2.8
PF 4.17	X	D	C	model display (PF 2.3) + data-based control (PF 2.2)	Côté et al. (2008)	o
PF 4.18	C	D	C	monitored required behaviour = information display (PF 2.6) + required behaviour (PF 1.2)	Côté et al. (2008)	+
PF 4.19	B	D	C	monitored commanded behaviour = information display (PF 2.6) + commanded behaviour (PF 2.8)	Côté et al. (2008)	+
PF 4.20	X	X	D	(interchange) PF 4.8	-	PF 4.8
PF 4.21	C	X	D	(interchange) PF 4.9	-	PF 4.9
PF 4.22	B	X	D	(interchange) PF 4.10	-	PF 4.10
PF 4.23	X	C	D	(interchange) PF 4.17	-	PF 4.17
PF 4.24	C	C	D	(interchange) PF 4.18	-	PF 4.18
PF 4.25	B	C	D	(interchange) PF 4.19	-	PF 4.19

Three domains, two constrained IV

Problem frames with three problem domains (two constrained, part 3):

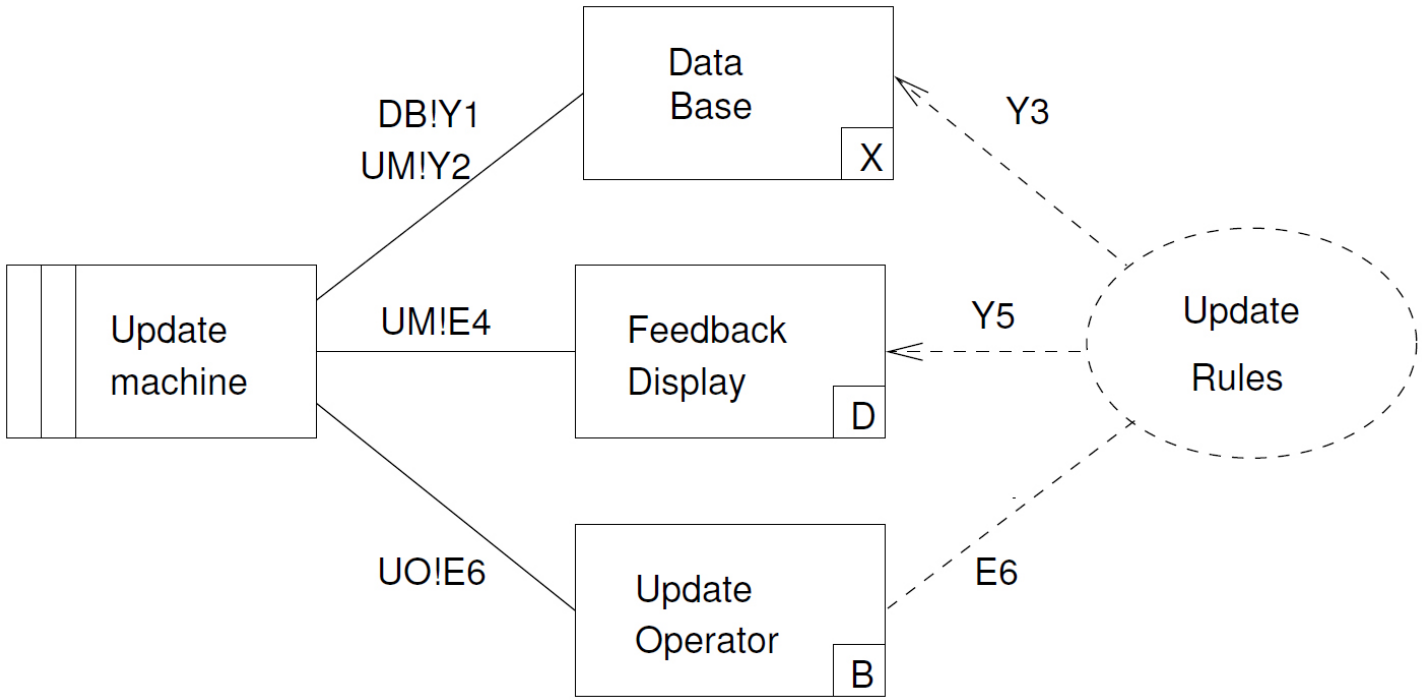
No.	Ref. dom. type (rdt?)	Constr. dom. 1 type (cdt1?)	Constr. dom. 2 type (cdt2?)	Comment	Author	Relevance
PF 4.26	X	D	D	= data-based control (merge)	Côté et al. (2008)	PF 2.2
PF 4.27	C	D	D	= information display (merge)	Jackson (2001)	PF 2.6
PF 4.28	B	D	D	= commanded display (merge)	Jackson (2001)	PF 2.9
PF 4.29	*	B	*	*Biddable cannot be constrained*	-	n/a
PF 4.30	*	*	B	*Biddable cannot be constrained*	-	n/a

Three domains, two constrained V

Update - Informal Description

An operator should be able to update or add information to a database. The database is changed, and feedback is given on a display.

Frame Diagram



A UML metamodel for problem frames

Approach

- **Formal metamodel** for problem frames
- **UML class model** and **OCL** (Object Constraint Language) integrity conditions

Benefits

- **Unambiguous comprehension** of the problem frames approach
- Possibility to **verify** that a frame is valid according to the metamodel
- Basis for **tool support**

A UML profile for problem frames I

UML Profiles (UML Revision Task Forces (2009))

- Provide a mechanism that allows metaclasses from existing metamodels to be extended to adapt them for different purposes
- This includes the ability to tailor the UML metamodel for different platforms or domains

A UML profile for problem frames II

Approach

- Definition of a **UML profile** (based on the metamodel) that allows to represent problem frames and problem diagrams in UML
- In this profile, **stereotypes** are defined which extend a UML meta-class from the UML meta-model (such as Association or Class)
- Additional, integrity conditions are defined and expressed as **OCL constraints**

A UML profile for problem frames III

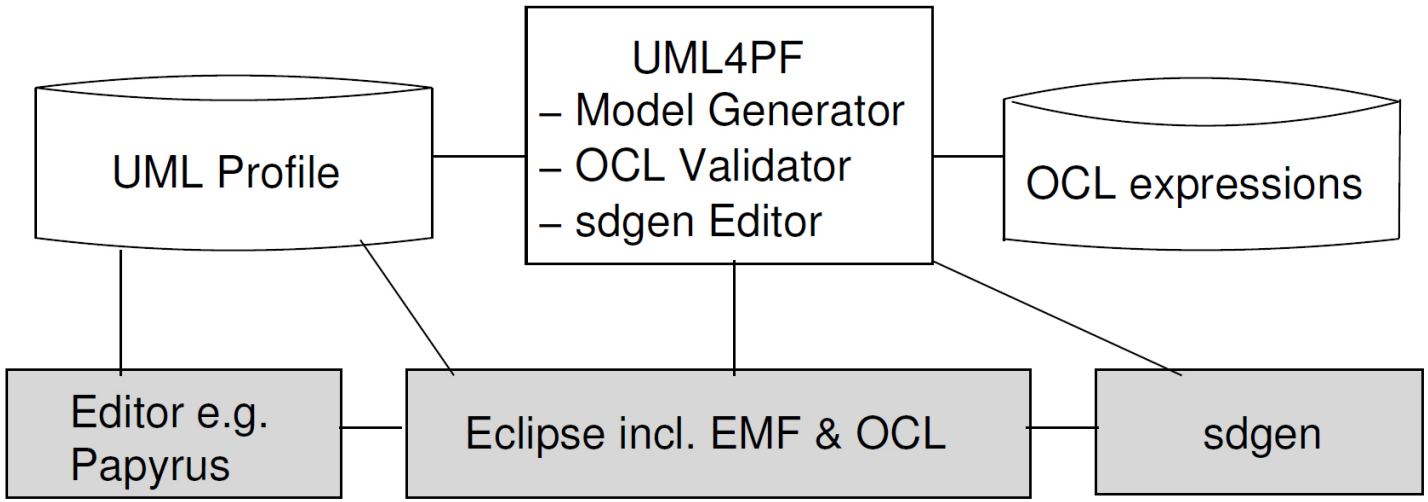
Approach

Provide tool support for

- pattern- and model-based software engineering
- starting with requirements engineering
- covering also subsequent phases, e.g., architectural design
- extension for dependability
- extension for software evolution
- extension for performance
- ...

UML4PF: tool concept

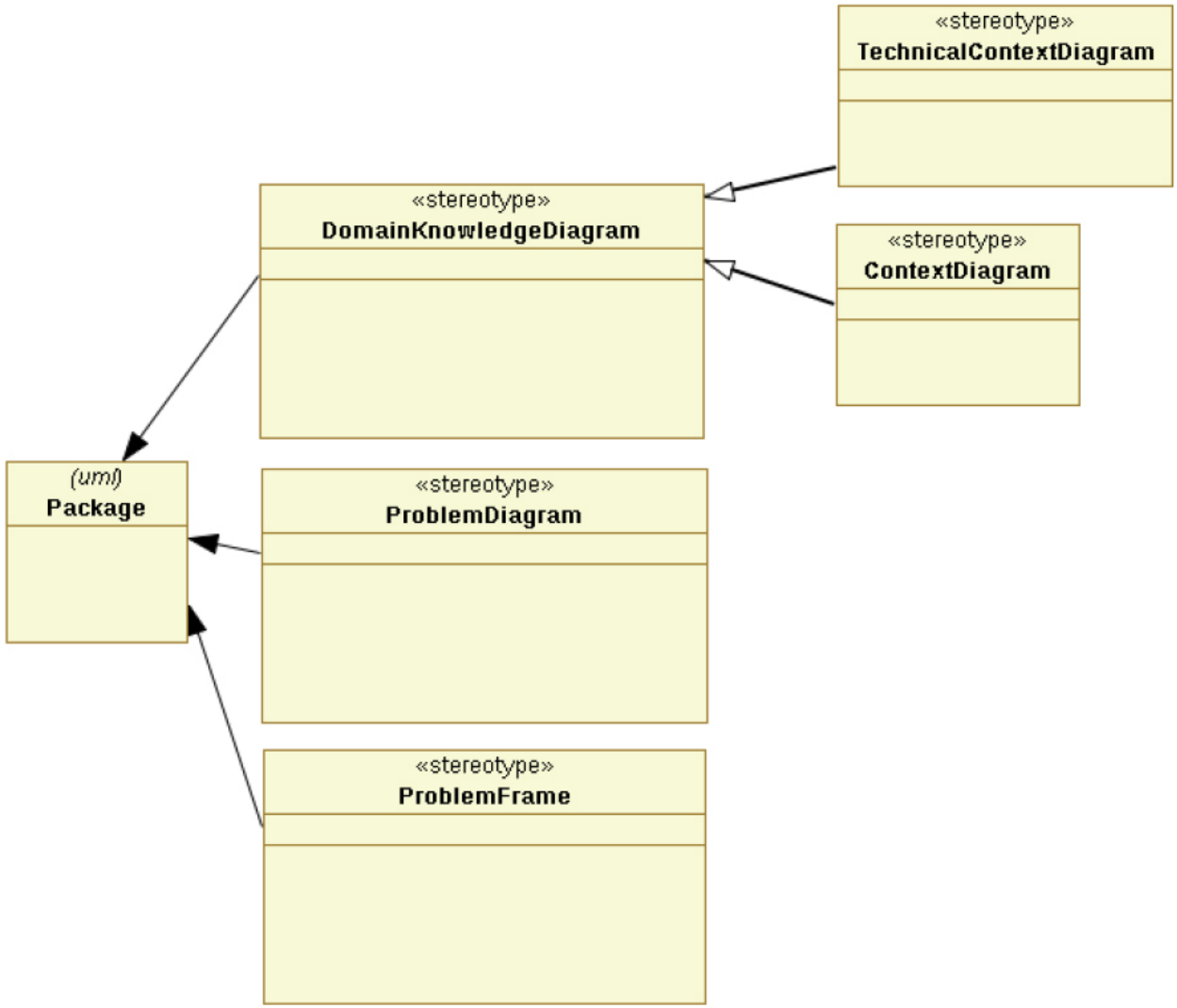
- Basis: Eclipse, with plug-ins EMF and OCL
- UML-profile for problem frames, extending the EMF meta-model, stored in XML-format
- OCL constraints, stored in XML format
- EMF-based editor for graphical representation of the different diagram types: Papyrus
- UML4PF: additional windows in Eclipse to edit requirements, easy-to-use user interface



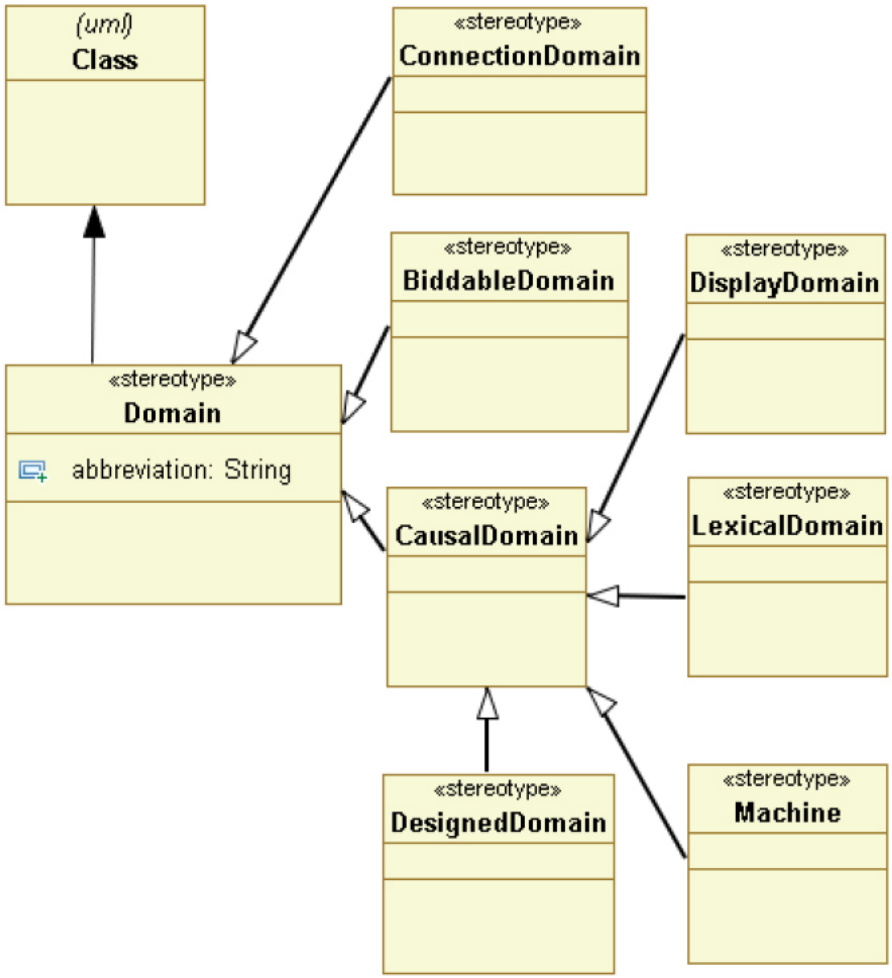
UML4PF: tool functionality

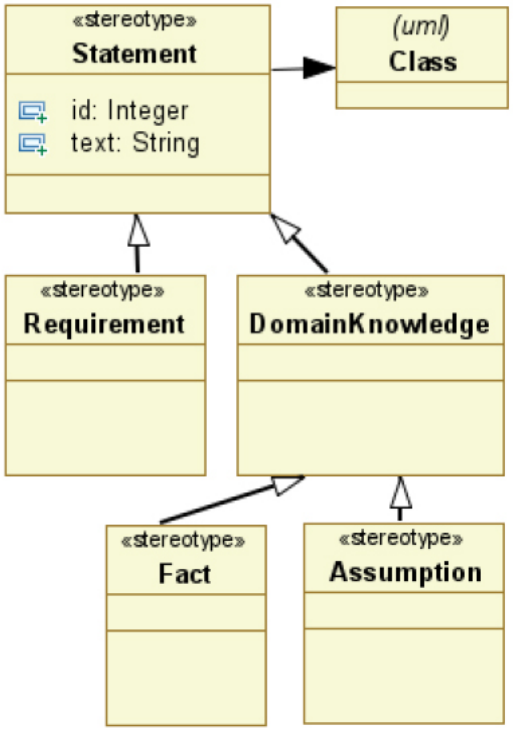
- automatically generate certain model elements, e.g., observed and controlled interfaces from association names
- check validity of single models
- check coherence between different models
- returns location of invalid parts of models
- support requirements tracing

Diagram Types



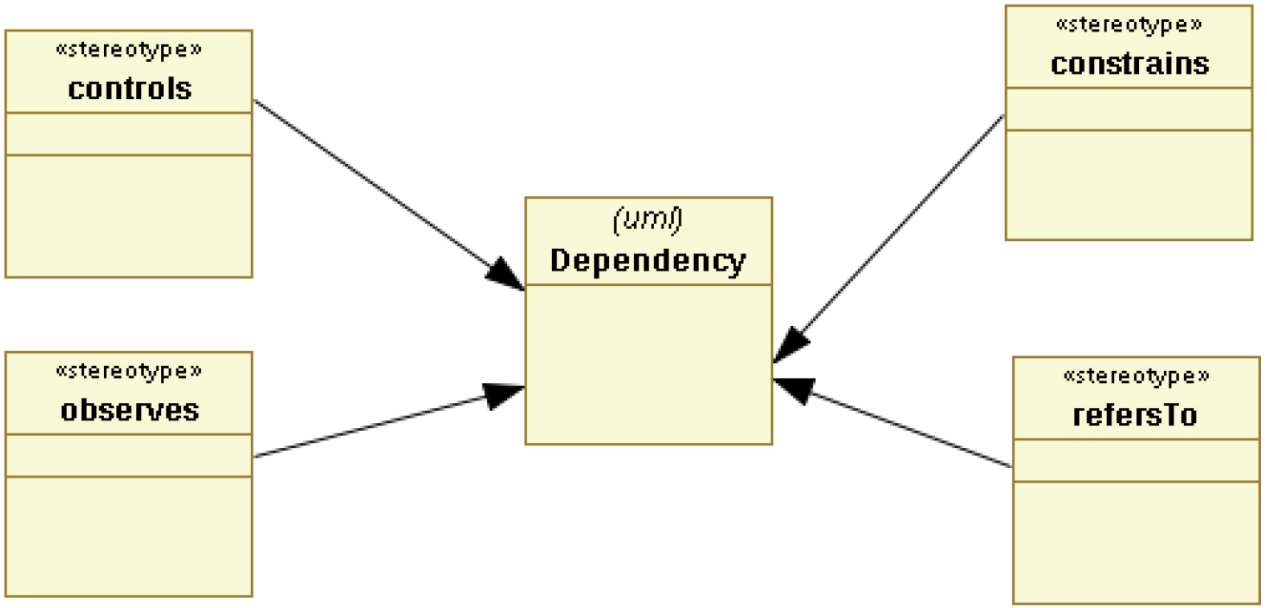
Domains and Statements



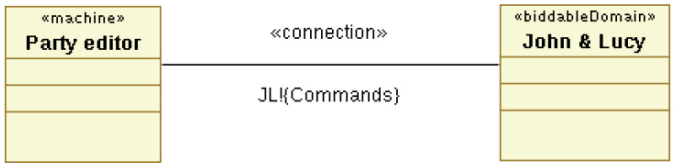


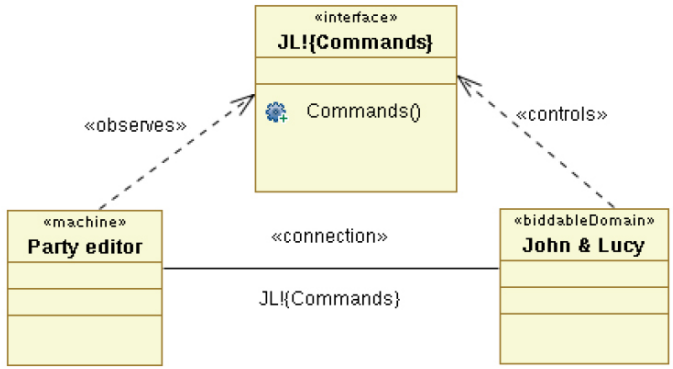
Dependencies: Requirements

- A requirement refers to some domains and constraints at least one domain
- This is expressed using the stereotypes `<<refersTo>>` and `<<constrains>>`, which extend the UML meta-class `Dependency`



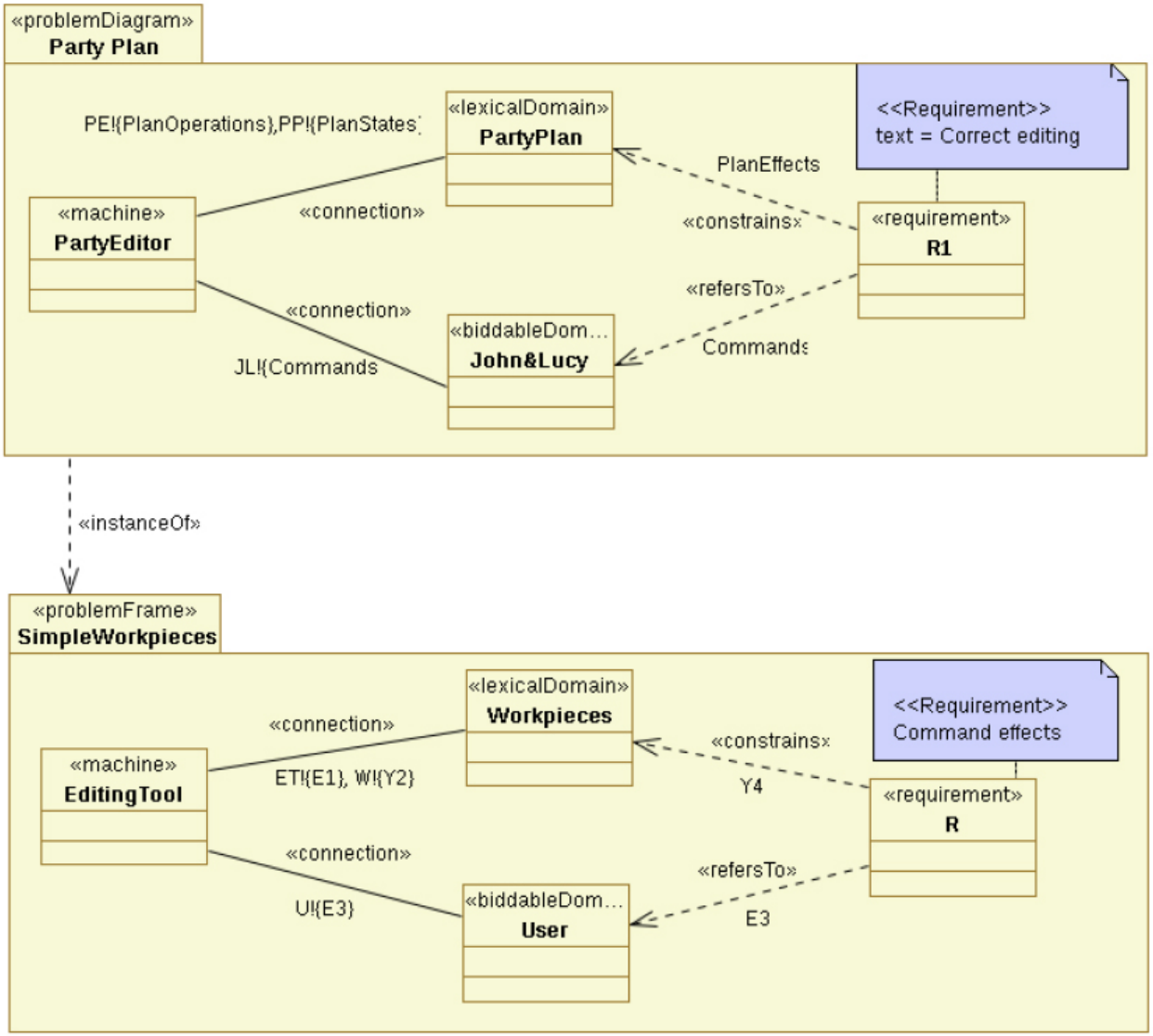
Dependencies: Shared phenomena





- Use of associations to describe the interfaces in problem diagrams (stereotype `<<connection>>`)
- Phenomena can be represented as operations in UML interface classes
- Connection `<<connection>>` can be transformed into an interface class **controlled** by a domain and **observed** by other domains
- This transformation is done automatically by UML4PF

Example: Simple Workpieces



OCL integrity conditions: overview

- The UML profile is complemented by a (growing) set of **OCL constraints**
- OCL is part of UML
- OCL allows to describe constraints on object-oriented modelling artifacts such as class models
- **Invariants** are constraints that must always be fulfilled
- Two classes of constraints:
  - The first class enables to check the integrity and consistency of **single diagrams**
  - The second class allows to check the consistency of **different type of diagrams**

What have we learned? I

- Simple problems that occur during problem decomposition should belong to known problem frames, which have common solution methods.
- These kinds of problem classes are characterized through *problem frames*. Problem frames contain patterns for the domains involved and their shared phenomena. They also define which domains the requirements refer to and which domains the restrict.
- In order to solve a subproblem with a method that is associated with a problem frame, the subproblem needs to be "fitted" to the frame by instantiating the frame diagram.

What have we learned? II

- Problem frames can be carried over to UML.
- Thus, their application can be tool-supported with existing UML tools.
- Problem frames can also be defined for non-functional requirements.

References I

Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3rd ed.). Addison-Wesley Professional.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional.

Jackson, M. (2001). *Problem Frames. Analyzing & Structuring Software Development Problems*. Addison Wesley.