Softwaretechnik / Software-Engineering

Lecture 14: Architecture and Design Patterns

2015-07-02

Prof. Dr. Andreas Podelski, **Dr. Bernd Westphal**

Albert-Ludwigs-Universität Freiburg, Germany

Contents of the Block "Design"

(i) **Introduction and Vocabulary**

(ii) **Principles of Design**

a) modularity
b) separation of concerns
c) information hiding and data encapsulation
d) abstract data types, object orientation

(iii) **Software Modelling**

a) views and viewpoints, the 4+1 view
b) model-driven/based software engineering
c) Unified Modelling Language (UML)
d) **modelling structure**

1. (simplified) class diagrams
2. (simplified) object diagrams
3. (simplified) object constraint logic (OCL)

e) **modelling behaviour**

1. communicating finite automata
2. Uppaal query language
3. basic state-machines
4. an outlook on hierarchical state-machines

(iv) **Design Patterns**

Introduction	L 1:	20.4., Mo
	T 1:	23.4., Do
Development Process, Metrics	L 2:	27.4., Mo
	L 3:	30.4., Do
	L 4:	4.5., Mo
	T 2:	7.5., Do
	L 5:	11.5., Mo
Requirements Engineering	-	14.5., Do
	L 6:	18.5., Mo
	L 7:	21.5., Do
	-	25.5., Mo
	-	28.5., Do
	T 3:	1.6., Mo
	-	4.6., Do
	L 8:	8.6., Mo
	L 9:	11.6., Do
	L 10:	15.6., Mo
	T 4:	18.6., Do
Architecture & Design, Software Modelling	L 11:	22.6., Mo
	L 12:	25.6., Do
	L 13:	29.6., Mo
	L 14:	2.7., Do
	T 5:	6.7., Mo
Quality Assurance	L 15:	9.7., Do
	L 16:	13.7., Mo
Invited Talks	L 17:	16.7., Do
	T 6:	20.7., Mo
Wrap-Up	L 18:	23.7., Do

Contents & Goals

Last Lecture:

- Networks of CFA, Tool Demo (recording will be reconstructed), Implementable CFA

This Lecture:

- **Educational Objectives:** Capabilities for following tasks/questions.
  - What is the relation between greedy and standard semantics?
  - What is an Uppaal Query for, e.g., "location ℓ is reachable"?
  - What's the difference between CFA and UML State-Machines?
  - Can each network of UML State-Machines be encoded in CFA?
  - Explain an example of an architecture (design) pattern.
  - What is "software entropy"?

- **Content:**
  - Implementable CFA Cont'd
  - Uppaal Query Language
  - UML State-Machines
  - Architecture and Design Patterns (with examples)

Implementing CFA Cont'd

Recall: Implementable CFA

- Let each automaton in the network $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ be marked as either **environment** or **controller**.

  We call $\mathcal{C}$ **implementable** if and only if, for each **controller** $\mathcal{A}$ in $\mathcal{C}$,

  (i) $\mathcal{A}$ is deterministic,

  (ii) $\mathcal{A}$ reads/writes only its local variables,
  may also read variables written by **environment** automata,
  but only in modification vectors of edges with input synchronisation,

  (iii) $\mathcal{A}$ is **locally deadlock-free**, i.e. enabled edges with output-actions are not blocked forever.

- The communicating finite automaton $\mathcal{A} = (L, B, V, E, \ell_{ini})$ is called **deterministic** if and only if
  - for each location ℓ,
    - **either** all edges with ℓ as source location have pairwise different **input actions**,
    - **or** there is no edge with an input action starting at ℓ,
      and all edges starting at ℓ have pairwise (logically) disjoint guards.

- **Note**: implementable (i) and (ii) can be checked syntactically.

  Property (iii) is a property of the whole network.

  Can be checked with Uppaal:

$$(\mathcal{A}.\ell \wedge \varphi) \longrightarrow (\mathcal{A}.\ell')$$

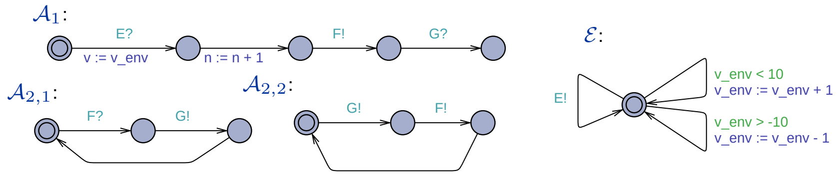
  for each edge $(\ell, \alpha, \varphi, \vec{r}, \ell')$ of $\mathcal{A}$.

Recall: Greedy CFA Semantics

- **Greedy** semantics:
  - each input synchronisation transition (plus: system start) of automaton $\mathcal{A}$ is followed by a maximal sequence of internal transitions or output transitions of $\mathcal{A}$.
  - **Maximal**: cannot be extended by an internal transition.

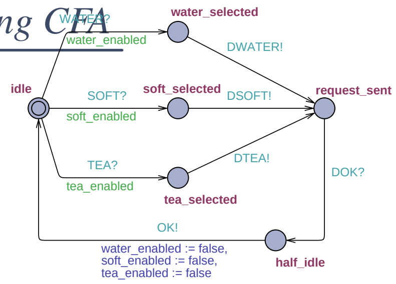
  There may still be interleaving of the internal transitions, but (by forbidding shared variables for controllers) cannot be observed outside of an automaton.

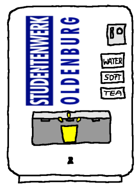
Example:



- $\mathcal{A}_1$ is implementable in $\mathcal{C}(\mathcal{A}_1, \mathcal{A}_{2,1}, \mathcal{E})$ (environment: only $\mathcal{E}$)
  - deterministic: ✔,
  - only local variables, environment variables with input: ✔,
  - locally deadlock-free: ✔.
- $\mathcal{A}_1$ is **not** implementable in $\mathcal{C}(\mathcal{A}_1, \mathcal{A}_{2,2}, \mathcal{E})$.

Recall: Implementing CFA






```
st : { idle, wsel, ssel, tsel, reqs, half };

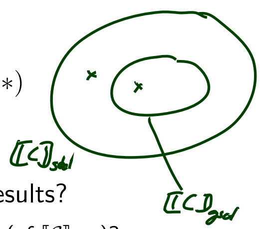
take_event( E : { TAU, WATER, SOFT, TEA, ...  } ) {
  bool stable = 1;
  switch (st) {
    case idle :
      switch (E) {
        case WATER :
          if (water_enabled) { st := wsel; stable := 0; }
          ;;
        case SOFT :
          ...
      }
    case wsel:
      switch (E) {
        case TAU :
          send_DWATER(); st := reqs;
          ;;
} }
```


Model vs. Implementation

- Now an implementable model $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ has **two semantics**:
  - $[\![\mathcal{C}]\!]_{std}$ — standard semantics.
  - $[\![\mathcal{C}]\!]_{grd}$ — greedy semantics.

- Are they **related** in any way? They are: $[\![\mathcal{C}]\!]_{std} \supseteq [\![\mathcal{C}]\!]_{grd}$. $(*)$

  Exercise: prove $(*)$.



- What effect does this insight have on Uppaal verification results?
  - If there **is an error** in $[\![\mathcal{C}]\!]_{std}$, will it be in a correct implementation (of $[\![\mathcal{C}]\!]_{grd}$)?
    Not necessarily.
  - If there **is no error** in $[\![\mathcal{C}]\!]_{std}$, will a correct implementation (of $[\![\mathcal{C}]\!]_{grd}$) be error-free?
    Yes, definitely.

		Uppaal verification	
		shows no error	reports error
impl. has error	yes	false negative ✓	true positive ✓
	no	true negative ✓	false positive

Uppaal Query Language

(Larsen et al., 1997; Behrmann et al., 2004)

The Uppaal Query Language

Consider $\mathcal{N} = \mathcal{C}(\mathcal{A}_1, \ldots, \mathcal{A}_n)$ over data variables V.

- **basic formula**:

$$atom ::= \mathcal{A}_i.\ell \mid \varphi \mid \texttt{deadlock}$$

  where $\ell \in L_i$ is a location and φ an expression over V.

- **configuration formulae**:

$$term ::= atom \mid \texttt{not}\ term \mid term_1\ \texttt{and}\ term_2$$

- **existential path formulae**: (“**exists finally**”, “**exists globally**”)

$$e\text{-}formula ::= \exists\Diamond\ term \mid \exists\Box\ term$$

- **universal path formulae**: (“**always finally**”, “**always globally**”, “**leads to**”)

$$a\text{-}formula ::= \forall\Diamond\ term \mid \forall\Box\ term \mid term_1\ \texttt{-->}\ term_2$$

- **formulae** (or **queries**):

$$F ::= e\text{-}formula \mid a\text{-}formula$$

Satisfaction of Uppaal Queries by Configurations

- The **satisfaction relation**

$$\langle \vec{\ell}, \nu \rangle \models F$$

between **configurations**

$$\langle \vec{\ell}, \nu \rangle = \langle (\ell_1, \dots, \ell_n), \nu \rangle$$

of a network $\mathcal{C}(\mathcal{A}_1, \dots, \mathcal{A}_n)$ and **formulae** F of the Uppaal logic is defined **inductively** as follows:

$(\ell_1, \ell_2, \dots, \ell_i, \dots, \ell_n)$

- $\langle \vec{\ell}, \nu \rangle \models$ `deadlock` iff $\langle \vec{\ell}, \nu \rangle$ is deadlock config.
- $\langle \vec{\ell}, \nu \rangle \models \mathcal{A}_i.\ell$ iff $\ell_i = \ell$
- $\langle \vec{\ell}, \nu \rangle \models \varphi$ iff $\nu \models \varphi$
- $\langle \vec{\ell}, \nu \rangle \models$ `not` *term* iff $\langle \vec{\ell}, \nu \rangle \not\models$ *term*
- $\langle \vec{\ell}, \nu \rangle \models term_1$ and $term_2$ iff $\langle \vec{\ell}, \nu \rangle \models term_1$ and $\langle \vec{\ell}, \nu \rangle \models term_2$

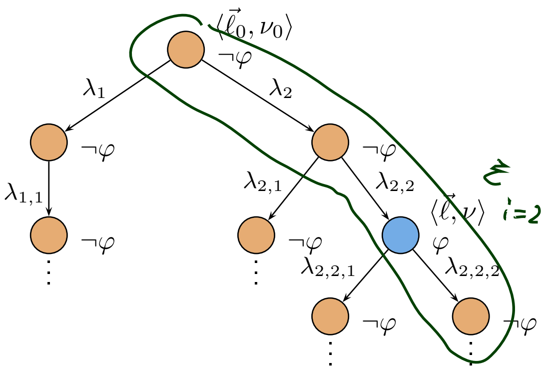
Satisfaction of Uppaal Queries by Configurations

Exists finally:

- $\langle \vec{\ell}_0, \nu_0 \rangle \models \exists\lozenge\, term$ iff $\exists$ path ξ of $\mathcal{C}$ starting in $\langle \vec{\ell}_0, \nu_0 \rangle$ $\exists\, i \in \mathbb{N}_0 \bullet \xi^i \models term$

"some configuration satisfying *term* is reachable" (from $\langle \vec{\ell}_0, \nu_0 \rangle$)

Example: $\exists\lozenge\, \varphi$



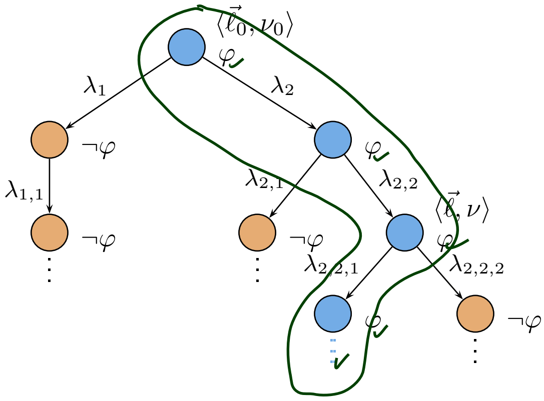
Satisfaction of Uppaal Queries by Configurations

Exists globally:

- $\langle \vec{\ell}_0, \nu_0 \rangle \models \exists\Box\, term$ iff $\exists$ path ξ of $\mathcal{C}$ starting in $\langle \vec{\ell}_0, \nu_0 \rangle$ $\forall\, i \in \mathbb{N}_0 \bullet \xi^i \models term$

"all configurations of some computation path satisfy $term$"

Example: $\exists\Box\,\varphi$



Satisfaction of Uppaal Queries by Configurations

- **Always globally**:
  - $\langle \vec{\ell}_0, \nu_0 \rangle \models \forall\Box\, term$ iff $\langle \vec{\ell}_0, \nu_0 \rangle \not\models \exists\Diamond\, \neg term$

    $\forall\Box\neg(pg \wedge tg)$ $\exists\Diamond\neg(\neg(pg \wedge tg))$ — $pg \wedge tg$
- **Always finally**:
  - $\langle \vec{\ell}_0, \nu_0 \rangle \models \forall\Diamond\, term$ iff $\langle \vec{\ell}_0, \nu_0 \rangle \not\models \exists\Box\, \neg term$

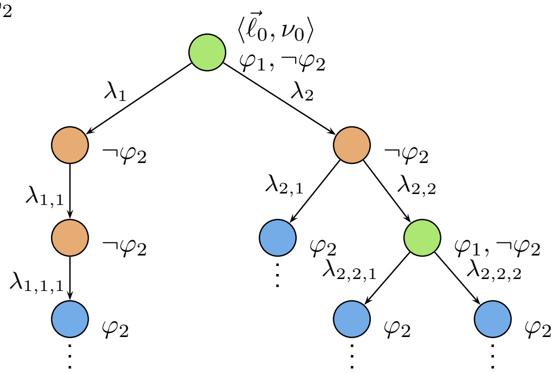
Satisfaction of Uppaal Queries by Configurations

Leads to:

- $\langle \vec{\ell}_0, \nu_0 \rangle \models term_1 \longrightarrow term_2$ iff $\forall$ path ξ of $\mathcal{N}$ starting in $\langle \vec{\ell}_0, \nu_0 \rangle$
  $\forall i \in \mathbb{N}_0 \bullet$
  $\xi^i \models term_1 \implies \xi^i \models \forall\Diamond\, term_2$

"on all paths, from each configuration satisfying $term_1$, a configuration satifying $term_2$ is reachable" (**response pattern**)

Example: $\varphi_1 \longrightarrow \varphi_2$



CFA Model-Checking

- **Network satisfies query**:
  - $\mathcal{C} \models F$ if and only if $C_{ini} \models F$.

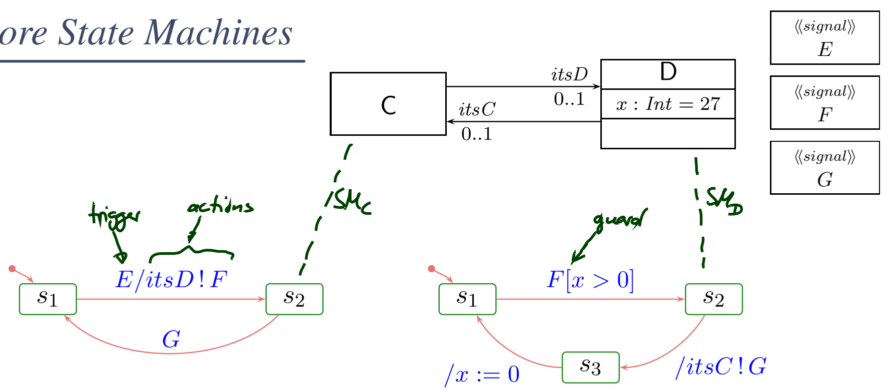
Definition. The **model-checking problem** for a network $\mathcal{C}$ of communicating finite automata and a query F is to decide whether

$$(\mathcal{C}, F) \in \models.$$

Proposition. The model-checking problem for communicating finite automata is decidable.

UML State Machines

UML Core State Machines

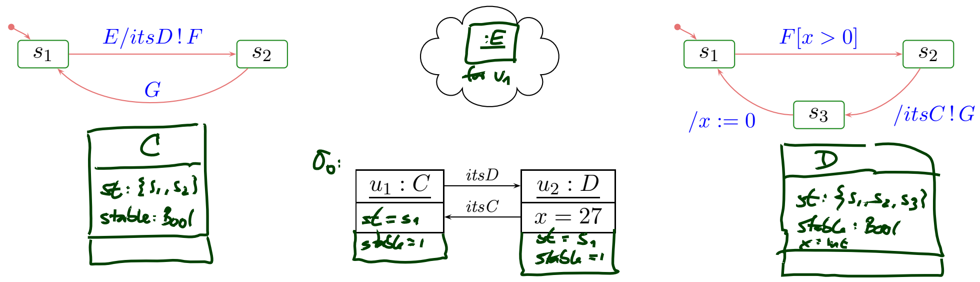


$$annot ::= [\underbrace{\langle event\rangle [\,.\,\langle event\rangle]^*}_{trigger} \quad [\,[\,\langle guard\rangle\,]\,] \quad [\,/\,\langle action\rangle\,]\,]$$

with

- $event \in \mathcal{E}$, (optional)
- $guard \in Expr_{\mathscr{S}}$ (default: *true*, assumed to be in $Expr_{\mathscr{S}}$)
- $action \in Act_{\mathscr{S}}$ (default: `skip`, assumed to be in $Act_{\mathscr{S}}$)

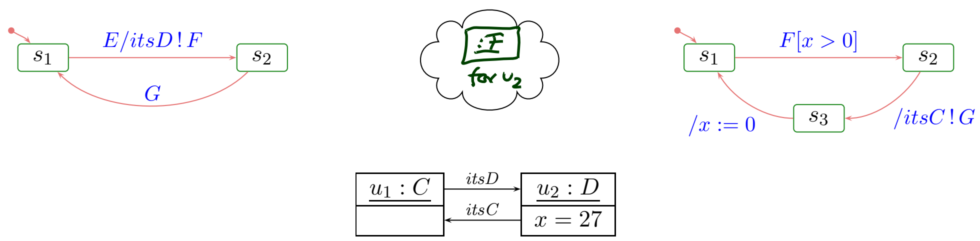
Event Pool and Run-To-Completion



σ_0:

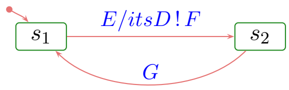
	u_1			u_2		
step	state	stable	x	state	stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1

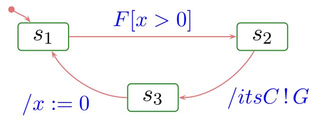
Event Pool and Run-To-Completion

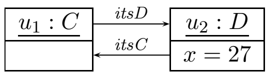


	u_1			u_2		
step	state	stable	x	state	stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2

Event Pool and Run-To-Completion

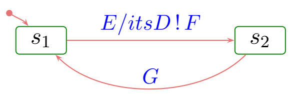


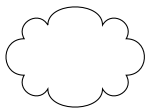


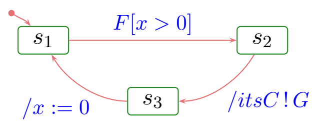


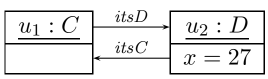
	u_1			u_2		
step	state	stable	x	state	stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2
2	s_2	1	27	s_2	0	
3	s_2	1	27	s_3	0	G ready for u_1

Event Pool and Run-To-Completion



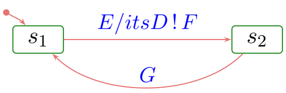


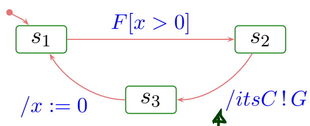


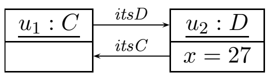


	u_1			u_2		
step	state	stable	x	state	stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2
2	s_2	1	27	s_2	0	
3	s_2	1	27	s_3	0	G ready for u_1
4.a	s_2	1	0	s_1	1	G ready for u_1
5.a	s_1	1	0	s_1	1	
4.b	s_1	1	27	s_3	0	

Event Pool and Run-To-Completion







what if trigger E here?
– nothing, computation stops

step	u_1 state	u_1 stable	x	u_2 state	u_2 stable	event pool
0	s_1	1	27	s_1	1	E ready for u_1
1	s_2	1	27	s_1	1	F ready for u_2
2	s_2	1	27	s_2	0	
3	s_2	1	27	s_3	0	G ready for u_1
4.a	s_2	1	0	s_1	1	G ready for u_1
5.a	s_1	1	0	s_1	1	
4.b	s_1	1	27	s_3	0	
5.b	s_1	1	0	s_1	1	

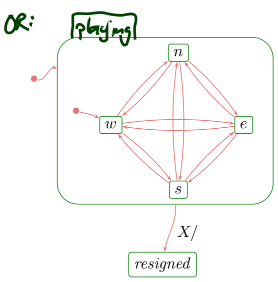
run-to-completion

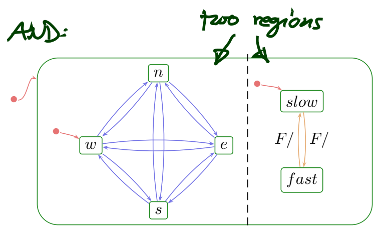
– if «signal, env» E: another E may come from the environment

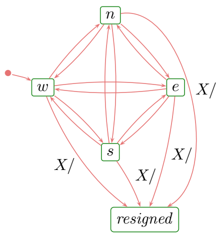
6 : —"— (G is discarded)

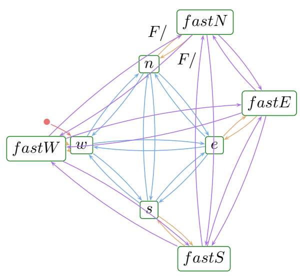
Composite (or Hierarchical) States

- OR-states, AND-states Harel (1987).
- Composite states are about **abbreviation**, **structuring**, and **avoiding redundancy**.

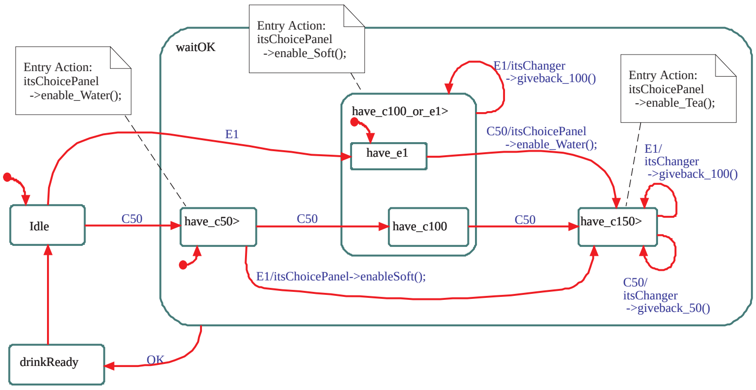


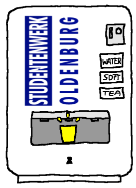


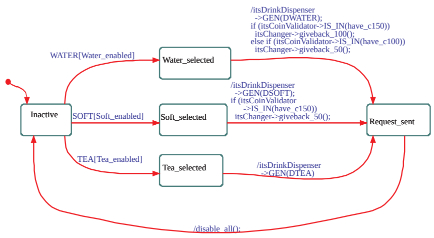


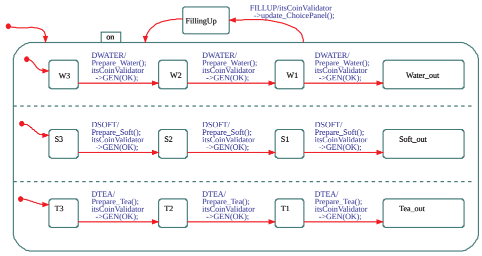


Example

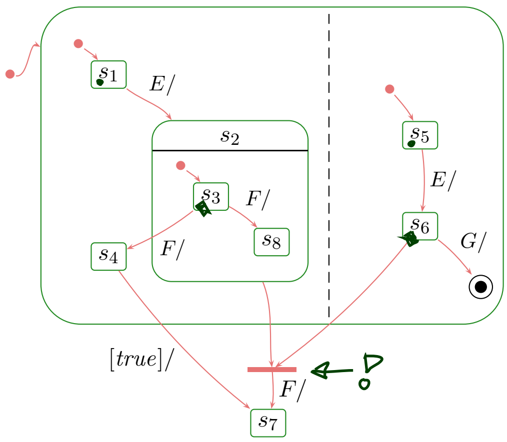






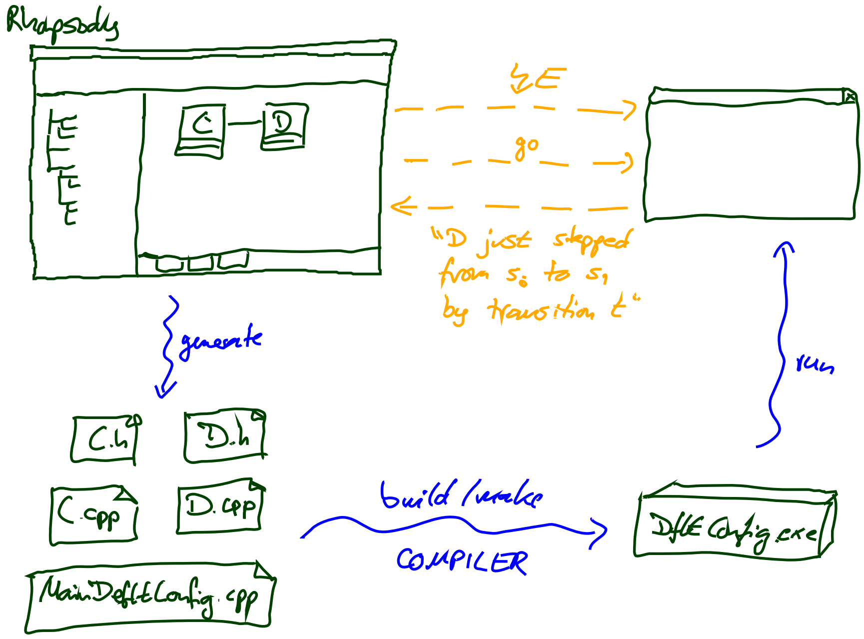


Would be Too Easy…



→ "**Software Design, Modelling, and Analysis with UML**" in the winter semester.

Rhapsody Architecture



UML Modes

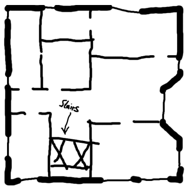
UML and the Pragmatic Attribute

Recall: definition "model" (Glinz, 2008, 425):

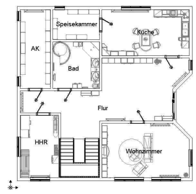
> [...] (iii) the **pragmatic attribute**, i.e. the model is built in a specific context for a specific purpose.

Examples for context/purpose:

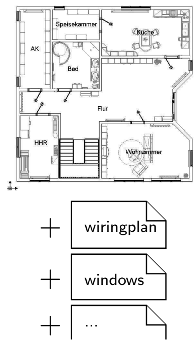
Floorplan as sketch:



Floorplan as blueprint:



Floorplan as program:



With UML it's the Same [http://martinfowler.com/bliki]

The last slide is inspired by **Martin Fowler**, who puts it like this:

> *"[...] people differ about what should be in the UML because there are* **differing fundamental views about what the UML should be**.
>
> *I came up with three primary classifications for thinking about the UML:*
>
> **UmlAsSketch**, **UmlAsBlueprint**, *and* **UmlAsProgrammingLanguage**.
>
> *([...] S. Mellor independently came up with the same classifications.)*
>
> *So when someone else's view of the UML seems rather different to yours, it may be because they use a different* **UmlMode** *to you."*

Claim:

- This not only applies to UML **as a language** (what should be in it etc.?),
- but at least as well to each individual UML **model**.

With UML it's a S...

The la...

Sketch

In this UmlMode developers use the UML to help communicate some aspects of a system. [...]

Sketches are also useful in documents, in which case the focus is communication ra- ther than completeness. [...]

The tools used for sketching are lightweight drawing tools and often people aren't too particular about keeping to every strict rule of the UML. Most UML diagrams shown in books, such as mine, are sketches. Their emphasis is on selective communication rather than complete specification. Hence my sound-bite "comprehensiveness is the enemy of comprehensibility"

Blueprint

[...] In forward engineering the idea is that blueprints are developed by a designer whose job is to build a detailed design for a programmer to code up. That design should be sufficiently complete that all design decisions are laid out and the programming should follow as a pretty straightforward activity that requires little thought. [...]

Blueprints require much more sophisticated tools than sketches in order to handle the details required for the task. [...]

Forward engineering tools support diagram drawing and back it up with a repository to hold the information. [...]

ProgrammingLanguage

If you can detail the UML enough, and provide semantics for everything you need in software, you can make the UML be your programming language.

Tools can take the UML diagrams you draw and compile them into executable code.

The promise of this is that UML is a higher level language and thus more productive than current programming languages.

The question, of course, is whether this promise is true.

I don't believe that graphical programming will succeed just because it's graphical. [...]

Claim:

- This
- but

UML-Mode of the Lecture: As Blueprint

- The "mode" fitting the lecture best is **AsBlueprint**.

Goal:

- be precise to **avoid misunderstandings**.
- allow formal **analysis of consistency/implication** on the **design level** — find errors early.

Yet we tried to be consistent with the (informal semantics) from the standard documents OMG (2007a,b) as far as possible.

Plus:

- Being precise also helps to work in mode **AsSketch**:
  Knowing "the real thing" should make it easier to
  (i) "see" which blueprint(s) the sketch is supposed to denote, and
  (ii) to ask meaningful questions to resolve ambiguities.

Architecture Patterns

Introduction

- Over decades of software engineering, many clever, proved and tested designs of solutions for particular problems emerged.
- **Question**: can we generalise, document and re-use these designs?
- **Goal**: "don't re-invent the wheel" / benefit from "clever", "proven and tested", "solution".

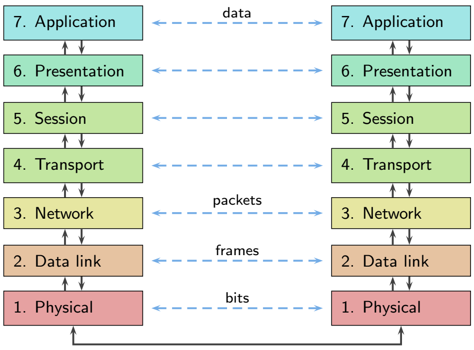
> **architectural pattern** — An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them. **Buschmann et al. (1996)**

- **Using** an architectural pattern
  - **implies** certain characteristics or properties of the software (construction, extendibility, communication, dependencies, etc.),
  - **determines** structures on a high level of the architecture, thus is typically a central and fundamental design decision.
- The information that (where, how, ...) a well-known architecture / design pattern is used in a given software can make comprehension and **maintenance** significantly easier.

Example: Layered Architectures

- (Züllighoven, 2005):
  - A layer whose components only interact with components of their direct neighbour layers is called **protocol-based** layer. A protocol-based layer hides all layers beneath it and defines a protocol which is (only) used by the layers directly above.

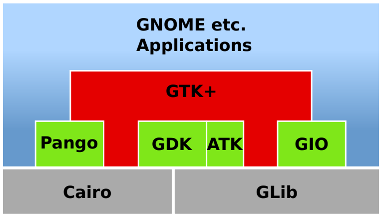
- **Example: The ISO/OSI reference model.**



Example: Layered Architectures Cont'd

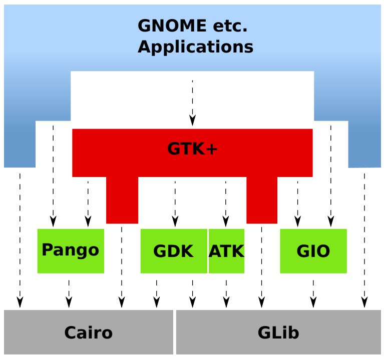
- **object-oriented layer**: interacts with layers directly and possibly further above and below.

- **Rules**: the components of a layer may use
  - **only** components of the protocol-based layer directly beneath,
  - **all** components of layers further beneath.



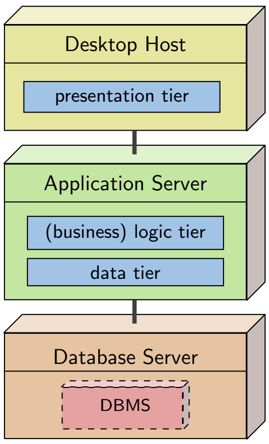
Example: Layered Architectures Cont'd

- **object-oriented layer**: interacts with layers directly and possibly further above and below.
- **Rules**: the components of a layer may use
  - **only** components of the protocol-based layer directly beneath,
  - **all** components of layers further beneath.



Example: Three-Tier Architecture

- **presentation layer**:
  user interface; presents information obtained from the logic layer to the user, controls interaction with the user, i.e. requests actions at the logic layer according to user inputs,
- **logic layer**:
  core system functionality; layer is designed without information about the presentation layer, may only read/write data according to data layer interface
- **data layer**:
  persistent data storage; hides information about how data is organised, read, and written, offers particular chunks of information in a form useful for the logic layer.



(Ludewig and Lichter, 2013)

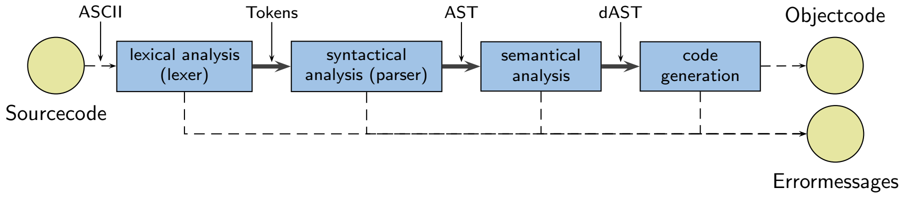
- **Examples**: Web-shop, business software (enterprise resource planning), etc.

Layered Architectures: Discussion

- **Advantages**:
  - protocol-based: only neighouring layers are coupled, i.e. components of these layers interact,
  - coupling is low, data usually encapsulated,
  - changes have local effect (only neighbouring layers affected),
  - protocol-based: distributed implementation often easy.
- **Disadvantages**:
  - performance (as usual), nowadays often not a problem.

Example: Pipe-Filter

Example: **Compiler**

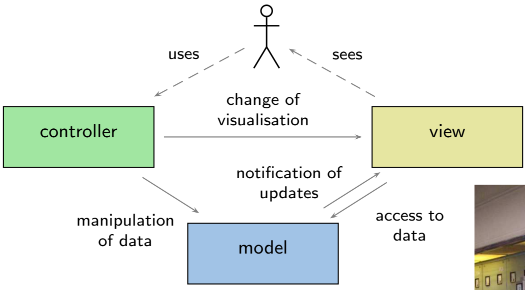


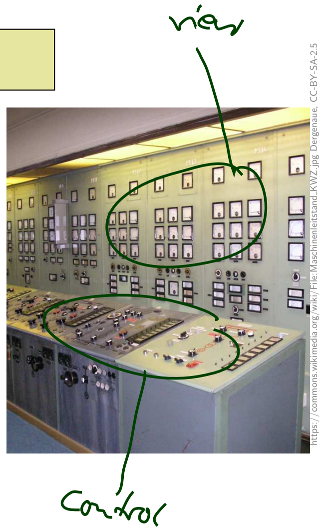
Example: **UNIX Pipes**

```
ls -l | grep Sarch.tex | awk '{ print $5 }'
```

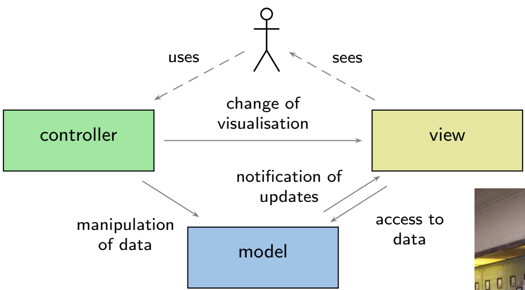
- **Disadvantages**:
  - if the filters use a common data exchange format, all filters may need changes if the format is changed, or need to employ (costly) conversions.
  - filters do not use global data, in particular not to handle error conditions.

Example: Model-View-Controller

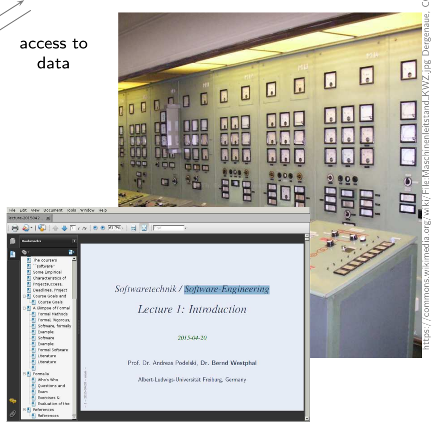




Example: Model-View-Controller



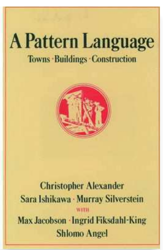
- **Advantages**:
  - one model can serve multiple view/controller pairs;
  - view/controller pairs can be added and removed at runtime;
  - model visualisation always up-to-date in all views;
  - distributed implementation (more or less) easily.
- **Disadvantages**:
  - if the view needs **a lot of data**, updating the view can be inefficient.

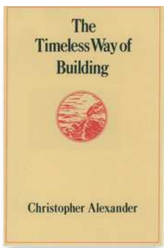


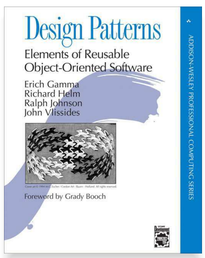
Design Patterns

Design Patterns

- In a sense the same as **architectural patterns**, but on a lower scale.
- Often traced back to (Alexander et al., 1977; Alexander, 1979).

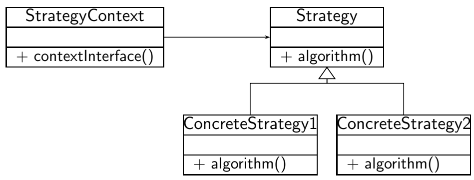




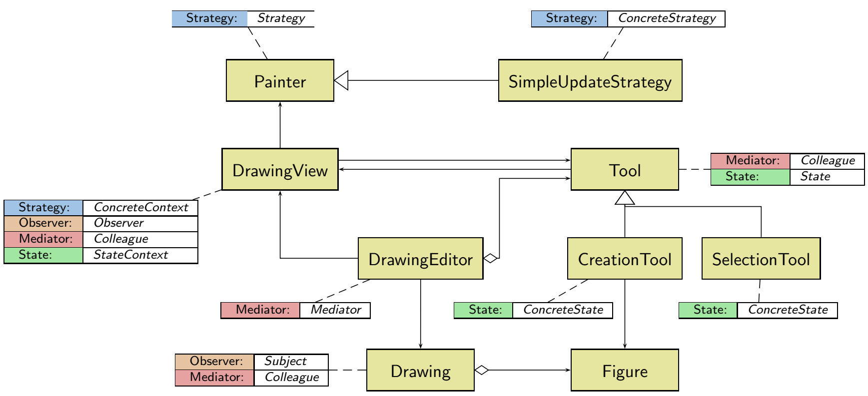


Design patterns ... are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design. **(Gamma et al., 1995)**

Example: Strategy

	Strategy
Problem	The only difference between similar classes is that they solve the same problem by different algorithms.
Solution	• Have one class **StrategyContext** with all common operations. • Another class **Strategy** provides signatures for all operations to be implemented differently. • From Strategy derive one sub-class **ConcreteStrategy** for each implementation alternative. • StrategyContext uses concrete Strategy-objects to execute the different implementations via delegation.
Structure	

Example: Pattern Usage and Documentation



Pattern usage in JHotDraw framework (JHotDraw, 2007) (Diagram: (Ludewig and Lichter, 2013))

Example: Singleton and Memento

	Singleton
Problem	Of one class, exactly one instance should exist in the system.
Example	Print spooler.

	Memento
Problem	The state of an object needs to be archived in a way that allows to re-construct this state without violating the principle of data encapsulation.
Example	Undo mechanism.

Example: Mediator, Observer, and State

	Mediator
Problem	Objects interacting in a complex way should only be loosely coupled and be easily exchangeable.
Example	Appearance and state of different means of interaction (menus, buttons, input fields) in a graphical user interface (GUI) should be consistent in each interaction state.

	Observer
Problem	Multiple objects need to adjust their state if one particular other object is changed.
Example	All GUI object displaying a file system need to change if files are added or removed.

	State
Problem	The behaviour of an object depends on its (internal) state.
Example	The effect of pressing the room ventilation button depends (among others?) on whether the ventilation is on or off.

Meta Design Pattern: Inversion of Control

- "**don't call us, we'll call you**"
- **Classical** (small) embedded controller software:
  - ```
    while (true) {
       // read inputs
       // compute updates
       // write outputs
    }
    ```
- **User interfaces**, for example:
  - define `button_callback();`
  - register method with UI-framework (→ later),
  - whenever button is pressed (handled by UI-framework), `button_callback()` is called and does its magic.
- Also found in **MVC** and **observer** patterns: model notifies view, subject notifies observer.

Design Patterns: Discussion

- "The development of design patterns is considered to be one of the most important innovations of software engineering in recent years." (Ludewig and Lichter, 2013)
- **Advantages**:
  - **(Re-)use** the experience of others and employ well-proven solutions.
  - Can improve on **quality criteria** like changeability or re-use.
  - Provide a **vocabulary** for the design process, thus facilitates documentation of architectures and discussions about architecture.
  - Can be combined in a flexible way, one class in a particular architecture can correspond to roles of multiple patterns.
  - Helps teaching software design.
- **Disadvantages**:
  - Using a pattern is not a value as such — using too much global data cannot be justified by "but it's the pattern Singleton".
  - **Again**: reading is easy, writing need not be.

    Here: Understanding abstract descriptions of design patterns or their use in existing software may be easy — using design patterns appropriately in new designs requires (**surprise, surprise**) experience.
    ```

Libraries and Frameworks

- **(Class) Library**: a collection of operations or classes offering generally usable functionality in a re-usable way.

  Examples:

  - `libc` — standard C library (is in particular abstraction layer for operating system functions),
  - `GMP` — GNU multi-precision library, cf. Lecture 6.
  - `libz` — compress data.
  - `libxml` — read (and validate) XML file, provide DOM tree.

- **Framework**: an architecture consists of class hierarchies which determine a generic solution for similar problems in a particular context.

  - **Example**: Android Application Framework

Libraries and Frameworks

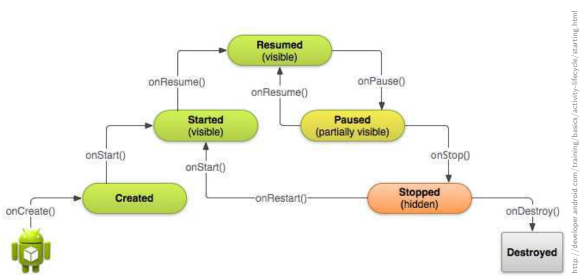
- **(Class) Library**: a collection of operations or classes offering generally usable functionality in a re-usable way.

  Examples:

  - `libc` — standard C library (is in particular abstraction layer for operating system functions),
  - `GMP` — GNU multi-precision library, cf. Lecture 6.
  - `libz` — compress data.
  - `libxml` — read (and validate) XML file, provide DOM tree.

- **Framework**: an architecture consists of class hierarchies which determine a generic solution for similar problems in a particular context.

  - **Example**: Android Application Framework



Libraries and Frameworks

- **(Class) Library**: a collection of operations or classes offering generally usable functionality in a re-usable way.

  Examples:
  - `libc` — standard C library (is in particular abstraction layer for operating system functions),
  - `GMP` — GNU multi-precision library, cf. Lecture 6.
  - `libz` — compress data.
  - `libxml` — read (and validate) XML file, provide DOM tree.

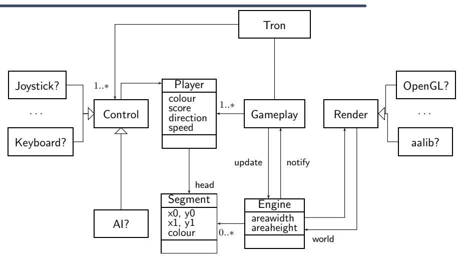
- **Framework**: an architecture consists of class hierarchies which determine a generic solution for similar problems in a particular context.
  - **Example**: Android Application Framework

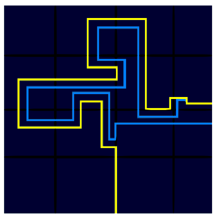
- The difference lies in **flow-of-control**:
  library modules are called from user code, frameworks call user code.

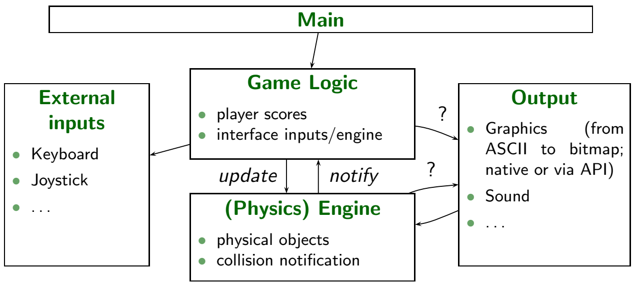
- **Product line**: parameterised design/code
  ("all turn indicators are equal, turn indicators in premium cars are more equal").

- For some application domains, there are **reference architectures** (games, compilers).

Reference Architecture Example: Games







Quality Criteria on Architectures

- **testability**
  - architecture design should keep testing (or formal verification) in mind (**buzzword** "design for verification"),
  - high locality of design units may make testing significantly easier (module testing),
  - particular testing interfaces may improve testability (e.g. allow injection of user input not only via GUI, or provide particular log output for tests).
- **changeability**, **maintainability**
  - most systems that are used need to be changed or maintained, in particular when requirements change,
  - **risk assessment**: parts of the system with high probability for changes should be designed such that changes are possible with acceptable effort (abstract, modularise, encapsulate),
- **portability**
  - systems with a long lifetime may need to be adapted to different platforms over time, infrastructure like databases may change,
  - **porting**: adaptation to different platform (OS, hardware, infrastructure).

- **Note**: a good design (model) is first of all supposed to **support the solution**, it need not be a good **domain model**.

Software Entropy

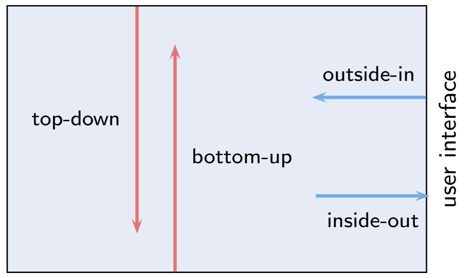
- **Lehman**'s Laws of Software Evolution (Lehman and Belady, 1985):
  (i) A program that is used **will be modified**.
  (ii) When a program is modified, its **complexity will increase**, provided that one does not actively work against this.
- **Software entropy** E (measure of disorder) Jacobson et al. (1992)

$$\text{claim: } \Delta E \sim E$$

  - "when designing a system with the intention of it being maintainable, we try to give it the lowest software entropy possible from the beginning."
  - Work against disorder: **re-factoring**
    (re-assign data and operations to modules, introduce new layers generalising old and new solutions, (automatically) check that intended interfaces are not bypassed, etc.)
- Proposal (Jacobson et al., 1992):
  - use "probability for change" as guideline in (architecture) design,
  - i.e. base design on a thorough analysis of problem and solution domain.

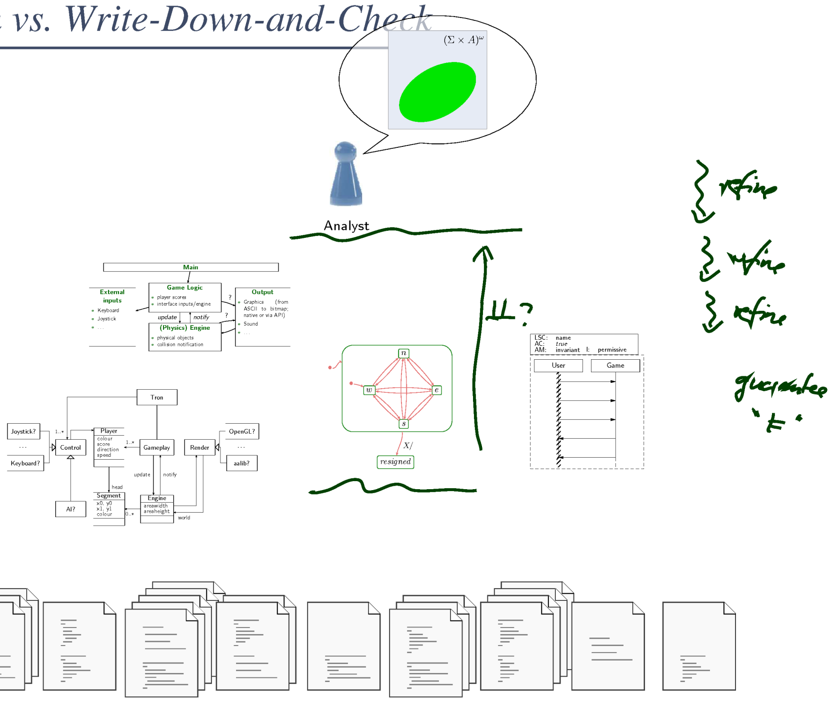
item	*probability for change*
Object from application [domain]	Low
Long-lived information structures	Low
Passive object's attribute	Medium
Sequences of behaviour	Medium
Interface with outside world	High
Functionality	High

Development Approaches



- **top-down** risk: needed functionality hard to realise on target platform.
- **bottom-up** risk: lower-level units do not "fit together".
- **inside-out** risk: user interface needed by customer hard to realise with existing system,
- **outside-in** risk: elegant system design not reflected nicely in (already fixed) UI.

Transform vs. Write-Down-and-Check



References

References

Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press.

Alexander, C., Ishikawa, S., and Silverstein, M. (1977). *A Pattern Language – Towns, Buildings, Construction*. Oxford University Press.

Behrmann, G., David, A., and Larsen, K. G. (2004). A tutorial on uppaal 2004-11-17. Technical report, Aalborg University, Denmark.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, E., and Stal, M. (1996). *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley & Sons.

Gamma, E., Helm, R., Johnsson, R., and Vlissides, J. (1995). *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Glinz, M. (2008). Modellierung in der Lehre an Hochschulen: Thesen und Erfahrungen. *Informatik Spektrum*, 31(5):425–434.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

Jacobson, I., Christerson, M., and Jonsson, P. (1992). *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley.

JHotDraw (2007). `http://www.jhotdraw.org`.

Larsen, K. G., Pettersson, P., and Yi, W. (1997). Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1):134–152.

Lehman, M. M. and Belady, L. (1985). *Program Evolution. Process of Software Change*. Academic Press.

Ludewig, J. and Lichter, H. (2013). *Software Engineering*. dpunkt.verlag, 3. edition.

OMG (2007a). Unified modeling language: Infrastructure, version 2.1.2. Technical Report formal/07-11-04.

OMG (2007b). Unified modeling language: Superstructure, version 2.1.2. Technical Report formal/07-11-02.

Züllighoven, H. (2005). *Object-Oriented Construction Handbook - Developing Application-Oriented Software with the Tools and Materials Approach*. dpunkt.verlag/Morgan Kaufmann.