Software Engineering

Prof. Federico Bergenti

Artificial Intelligence Laboratory

www.ailab.unipr.it/bergenti

Class Details

- Class schedule
  - Monday, 10:30am–12:30pm, room G
  - Tuesday, 10:30am–12:30pm, room G
  - Wednesday, 10:30am–12:30pm, room G
- Classes are live streamed and recorded
  - The course page on Elly (elly2023.smfi.unipr.it) mentions the live-streaming link, which is the same for all classes
  - The course page on Elly mentions the code to register to the team and access recordings
  - *It is strictly prohibited to download recordings*

Communications with the Teacher

- Few simple rules to communicate with the teacher
  - Communications and announcements from the teacher are posted via Elly
  - Meetings with the teacher are requested via e-mail
  - Before requesting a meeting, send an e-mail describing the the reasons for the meeting request
    - Clearly describe the problems with sufficient level of detail
    - Attach the exercises and/or the programs that originated the meeting request

federico.bergenti@unipr.it

Exam Details

- All exams are on the *current* course program
  - No real changes in the last 3 years
- Exam schedule (to be confirmed)
  - 3 sessions in January–March
  - 3 sessions in June–July
  - 1 session in August–September
- Exam sessions are composed of
  - A written exam
  - A lab exam
  - An oral exam (upon request of the teacher or the student)

Exam Details

- Note that
  - The lab exam follows a successful written exam
  - The oral exam follows a successful lab exam
  - The oral exam is upon explicit request of the teacher or the student
  - The oral exam is part of the process and it is not supposed to improve grades only
  - The written, lab, and oral exams must be passed in the same session
  - The written, lab, and oral exams cover the whole course program

Exam Details

- Exam enrollment
  - Enrollment is strictly requested to participate to an exam
  - Enrollment is performed using Esse3

unipr.esse3.cineca.it

  - Enrollment is possible only within the period associated with each session
  - Clearing the enrollment for an exam is possible only within the period associated with each session
- *Students are not supposed to enroll to all sessions*

Didactic Materials

- Classes are the official reference for the course
  - The slides used for classes will be available via Elly throughout the academic year
  - Class recordings will be available via Elly throughout the academic year
- Additional materials will be provided by the teacher via Elly
  - To expand and complement slides
  - To allow for deeper investigations on selected topics
- If help with Java is needed: *B. Eckel, Thinking in Java* (2nd edition, at least)

What is Software Engineering?

- Software engineering is about building large and complex software systems
  - The size and the complexity of the system call for the cooperation of several individuals, often organized in dispersed teams
- Software engineering is a direct consequence of the ever-increasing complexity of software systems

Possible Definitions

- Software engineering is…
  - …the systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software *(IEEE Systems and software engineering–Vocabulary)*
  - …the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software *(IEEE Standard Glossary of Software Engineering Terminology)*
  - …an engineering discipline that is concerned with all aspects of software production *(Ian Sommerville)*

Major Tasks (I)

- *(Software) Requirement analysis* is about the elicitation, analysis, specification, and validation of requirements for software systems
- *Software design* is about defining the architecture, components, interfaces, and other characteristics of a system, module, or component
- *Software development*, the main activity of software construction, is the combination of programming, verification, and debugging

Major Tasks (II)

- *Software testing* is an empirical investigation to provide stakeholders with information about the quality of the software product under test
- *Software maintenance* refers to the activities required to provide cost-effective support after shipping a software product
- *Software retirement* encompasses the activities required to provide cost-effective retirement of a software product

Historical Notes (I)

- Software engineering and related concepts were first discussed at the first *NATO Software Engineering Conference* in 1968

- New standpoints and keywords were proposed
  - Software crisis
  - Software reuse
  - Software engineering

Historical Notes (II)

- The practice of object-oriented technologies was energetically developed in the 1990s
  - (Almost-)Pure object-oriented programming languages and supporting tools
  - *UML* (*Unified Modeling Language*)
  - Design patterns

Historical Notes (II)

- From (around) 1997, innovations marked the software landscape
  - Web & e-commerce
  - Open-source software
- Java is
  - An (almost-)pure object-oriented language
  - A truly cross-platform runtime environment
  - A technology that nicely *fitted* the needs of the Web
  - A technology that the open-source software community *appreciated*

Concurrent Software Systems

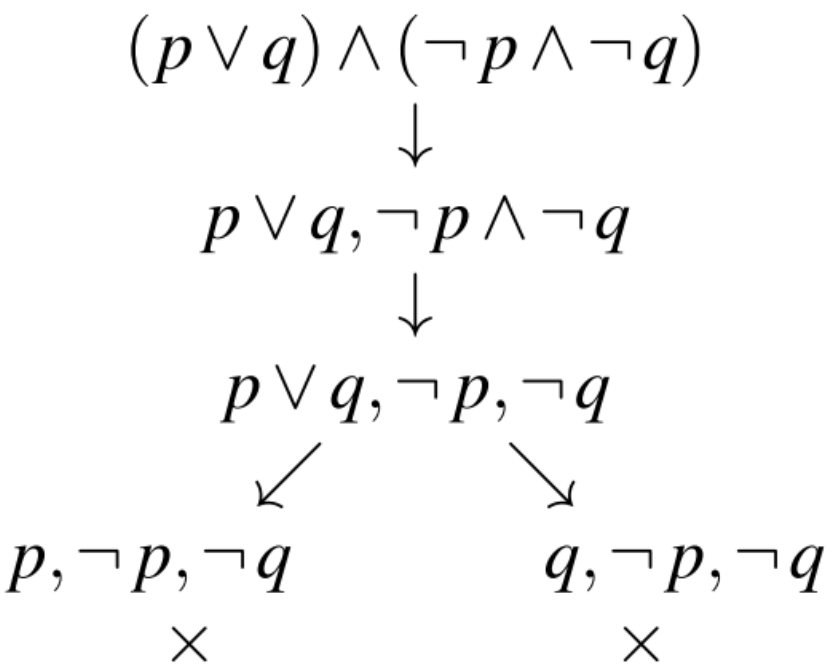
"Each philosopher had a room in which he could engage in his professional activity of thinking; there was also a common dining room, furnished with a circular table, surrounded by five chairs […] *When he was finished he would put down both his forks, get up from his chair, and continue thinking."*

C. A. R. Hoare, *Communicating Sequential Processes*, Prentice Hall, 1985

Propositional Logic and Software Engineering

- A simple tool to formally prove that interesting properties of studied (mathematical) objects hold

- In software engineering, an important tool to formally study the characteristics of software systems
  - To (mathematically) prove that relevant properties of studied systems hold



Logic Languages (I)

- A formal language is given in terms of
  - Its syntax, which is the set of *well-formed formulae*
  - Its semantics, which is an *interpretation* of well-formed formulae for a given *domain of discourse*
- A *logic language* is a formal language equipped with
  - *Axioms*, which are assumed truths
  - *Inference rules*, to obtain new truths from established truths (axioms or inferred truths)
- Logic languages can be used to prove *theorems* starting from axioms and inference rules

Logic Languages (II)

- Classical logic languages
  - *Propositional logic*, to reason using propositions and connectives ($p \wedge q$, $p \vee q$, $\neg p$, …)
  - *Predicate logic*, to reason using terms, predicates, connectives, quantifiers, and variables ($\forall x.man(x) \rightarrow mortal(x)$, …)
- Modal logic languages (for example)
  - *Epistemic logics*, to reason on beliefs and knowledge (*"A knows that P"*, *"B believes that P"*, …)
  - *Temporal logics*, to reason on temporal relationships (*"It is always true that P"*, *"It is true that P after Q"*, …)

Propositional Logic

- The language is based on *propositional symbols*
  - Propositional symbols are *atoms* (*p* stands for *"Alice loves Bob"*, *q* stands for *"Alice hate Cate"*, …)
  - Propositional symbols are meant to be *true* or *false*
- *Propositions* (or *propositional well-formed formulae*) are obtained combining propositional symbols with connectives ¬, $\wedge$, $\vee$, →, ≡
  - ¬*p* stands for *"Alice does not love Bob"*
  - *p*→*q* stands for *"if Alice loves Bob then Alice hates Cate"*
- A *literal* is a propositional symbol or its negation

Syntax (I)

- Given a countable set of propositional symbols $P \neq \emptyset$, the set *Prop*[*P*] of propositions over *P* is constructed as follows
  - Every propositional symbol is a proposition
  - $\top$ (*top*) and $\bot$ (*bottom*) are propositions (it is always assumed that $\top \notin P$ and $\bot \notin P$)
  - If *A* is a proposition, then $\neg A$ is a proposition
  - If *A* and *B* are proposition, then $A \wedge B$, $A \vee B$, $A \rightarrow B$, $A \equiv B$ are proposition
  - Nothing else is a proposition

Syntax (II)

- Syntactic conventions
  - Parentheses can be used to group propositions
    - Outer parentheses can be omitted
    - Square bracket and curly brace are also usable, but only parentheses are normally used
  - Binary connectives are *left associative*
    - $p \wedge q \wedge r$ shortens $((p \wedge q) \wedge r)$
  - The precedence of connectives follows the order $\neg, \wedge, \vee, \rightarrow, \equiv$
    - $p \rightarrow q \wedge r$ shortens $(p \rightarrow (q \wedge r))$

Semantics (I)

- The *semantics* of propositional logic is intended to reason on the truth and the falsity of propositions
- Propositional symbols must be assigned a truth value to ensure that the truth value of a complex proposition can be computed

- For example, does $p \wedge q$ hold?
  - The question has no unique answer
  - The answer depends on the truth values of p and q
    - If p is true and q is true, then $p \wedge q$ is true, otherwise $p \wedge q$ is false

Semantics (II)

- An *interpretation* is a function that assigns a truth value to each symbol in *P*
  - It is a total function over P whose codomain is an arbitrary binary set
  - For example
    $I : P \rightarrow \boldsymbol{B}$ $\quad\quad$ $\boldsymbol{B} = \{ F, T \}$
    where ***B*** is the chosen binary set composed of two symbols
- For example, if *P*={ *p*, *q*, *r* }, then there are *8* possible interpretations

	p	*q*	*r*
I_1	F	F	F
I_2	F	F	T
I_3	F	T	F
I_4	F	T	T
I_5	T	F	F
I_6	T	F	T
I_7	T	T	F
I_8	T	T	T

Semantics (III)

- Given an interpretation I over P, the interpretation G_I of an arbitrary proposition in $Prop[P]$ can be computed as follows
  - $G_I(A)=I(A)$ if $A \in P$
  - $G_I(\top)=\text{T}$ and $G_I(\bot)=\text{F}$
  - $G_I(\neg A)=\text{F}$ if $G_I(A)=\text{T}$, otherwise $G_I(\neg A)=\text{T}$
  - $G_I(A \wedge B)=\text{T}$ if $G_I(A)=\text{T}$ and $G_I(B)=\text{T}$, otherwise $G_I(A \wedge B)=\text{F}$
  - $G_I(A \vee B)=\text{F}$ if $G_I(A)=\text{F}$ and $G_I(B)=\text{F}$, otherwise $G_I(A \vee B)=\text{T}$
  - $G_I(A \rightarrow B)=\text{T}$ if and only if $G_I(A)=\text{F}$ or $G_I(B)=\text{T}$
  - $G_I(A \equiv B)=\text{T}$ if and only if $G_I(A)=G_I(B)=\text{T}$ or $G_I(A)=G_I(B)=\text{F}$

Models and Satisfiability (I)

- Given an interpretation *I* and a proposition *A*
  - $I \models A$ *(I satisfies A)* if and only if $G_I(A)=T$
  - $I \not\models A$ (*I does not satisfy A*) if and only if $G_I(A)=F$
- An interpretation *I* is a model for a proposition *A* if and only if $I \models A$
  - For example, if $A=(p \rightarrow q) \vee q$, and *I* is such that $I(p)=F$ and $I(q)=F$, then *I* is a model for *A*

- A proposition *A* is *satisfiable* if and only if there exists an interpretation *I* such $I \models A$

Models and Satisfiability (II)

- Given an interpretation I and a proposition A, the following rules can be used to *check* if $I \models A$
  - $I \models p$ if and only if $I(p)=\mathrm{T}$ and $p \in P$
  - $I \models \top$ and $I \not\models \bot$
  - $I \models \neg A$ if and only if $I \not\models A$
  - $I \models A \wedge B$ if and only if $I \models A$ and $I \models B$
  - $I \models A \vee B$ if and only if $I \models A$ or $I \models B$
  - $I \models A \rightarrow B$ if and only if $I \not\models A$ or $I \models B$
  - $I \models A \equiv B$ if and only if $I \models A$ and $I \models B$, or $I \not\models A$ and $I \not\models B$

Tautologies (I)

- A proposition A is *logically valid* (or, a *tautology*) if and only if $I \models A$ for any possible interpretation I
  - If A is a tautology, then $\models A$
  - All interpretations are models for A

- For example,
  - $p \rightarrow (p \vee q)$ is a tautology
  - $(p \rightarrow q) \vee q$ is satisfiable but it is not a tautology

Tautologies (II)

- For all propositions A and B, the following propositions are tautologies
  - $A \rightarrow A$
  - $A \rightarrow (B \rightarrow A)$
  - $\neg A \rightarrow (A \rightarrow B)$
  - $\bot \rightarrow B$ (*ex falso quodlibet*)
  - $A \vee \neg A$ (*tertium non datur*)
  - $\neg(A \wedge \neg A)$
  - $(A \rightarrow B) \rightarrow ((A \rightarrow \neg B) \rightarrow \neg A)$
  - $((A \rightarrow B) \rightarrow A) \rightarrow A$

Contradictions

- A proposition *A* is a *contradiction* (*unsatisfiable* proposition) if and only if no interpretation is a model for *A*
  - For example, $\neg(p \rightarrow (p \vee q))$

- A proposition *A* is
  - A tautology if and only if $\neg A$ is a contradiction
  - A contradiction if and only if $\neg A$ is a tautology

Logical Equivalences (I)

- Propositions A and B are logically equivalent ($A \leftrightarrow B$) if and only if $\models (A \equiv B)$
  - For any interpretation I, $I \models A$ if and only if $I \models B$

- The following are relevant logical equivalences
  - Law of excluded middle
    - $A \wedge \neg A \leftrightarrow \bot$
  - Commutative and associative laws
    - $A \wedge B \leftrightarrow B \wedge A$, $A \vee B \leftrightarrow B \vee A$,
      $A \wedge (B \wedge C) \leftrightarrow (A \wedge B) \wedge C$, $A \vee (B \vee C) \leftrightarrow (A \vee B) \vee C$
  - Distributive laws
    - $(A \vee (B \wedge C)) \leftrightarrow ((A \vee B) \wedge (A \vee C))$, $(A \wedge (B \vee C)) \leftrightarrow ((A \wedge B) \vee (A \wedge C))$

Logical Equivalences (II)

- A few other relevant logical equivalences
  - Law of double negation
    - $A \leftrightarrow \neg\neg A$
  - De Morgan's laws
    - $\neg(A \vee B) \leftrightarrow (\neg A \wedge \neg B)$, $\neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B)$
  - Law of contraposition
    - $(A \rightarrow B) \leftrightarrow (\neg B \rightarrow \neg A)$
  - Definability of logical connectives
    - $A \equiv B \leftrightarrow (A \rightarrow B) \wedge (B \rightarrow A)$
    - $A \rightarrow B \leftrightarrow \neg A \vee B \leftrightarrow \neg(A \wedge \neg B)$
    - $A \wedge B \leftrightarrow \neg(\neg A \vee \neg B) \leftrightarrow \neg(A \rightarrow \neg B)$
    - $A \vee B \leftrightarrow \neg(\neg A \wedge \neg B) \leftrightarrow \neg A \rightarrow B$

Logical Consequences (I)

- Interpretation *I* is a model for a set of propositions *S* if and only if it is a model for each and every proposition in *S*
- A set of propositions that has no models is said to be unsatisfiable
- Proposition *A* is a logical consequence of a set of propositions *S* ($S \models A$) if and only if any model *I* for *S* is also a model for *A*
  - For example, $\{ p \vee q, p \rightarrow r, q \rightarrow r \} \models r$
  - Note that $S \models A$ does not imply that all the models for *A* are also models for *S*

Logical Consequences (II)

- To check if $\{ A_1, \ldots, A_n \} \models B$
  - Write the *truth table* of $A_1 \wedge \cdots \wedge A_n \rightarrow$B and check if it is a tautology
  - Find an interpretation of $A_1 \wedge \cdots \wedge A_n \rightarrow$B that is not a model
  - Find an interpretation I such that $I \models A_1, \ldots, I \models A_n$ and check if I is a model for B
  - Obtain B starting from $A_1, \ldots, A_n$ using only a set of *sound* and *complete* inference rules
  - Apply the propositional tableau method, as discussed in the following slides

Negation Form

- A proposition is in *negation (normal) form* if
  - Only conjunctions, disjunctions, and negations are used in the proposition
  - Negations occur only in literals (remember: a literal is a propositional symbol or its negation)
- Any proposition can be turned into an equivalent proposition in negation form using
  - The law of double negation
  - De Morgan's laws
  - The definability of logical connectives

Conjunctive and Disjunctive Forms

- A proposition is
  - In *conjunctive (normal) form* if it is in negation form and it is structured as a conjunction of disjunctions of literals
  - In *disjunctive (normal) form* if it is in negation form and it is structured as a disjunction of conjunctions of literals

- Any proposition can be turned into an equivalent proposition in conjunctive or disjunctive form using
  - The distributive law
  - The law of double negation
  - De Morgan's laws
  - The definability of logical connectives

Propositional Tableaux (I)

- A *propositional (semantic) tableau* is a *tree* whose nodes are labelled with sets of propositions using the following rules until no rules are further applicable
  - An initial set of propositions in negation form labels the root of the tree
  - If a leaf of the tree is labeled with $X \cup \{ A \wedge B \}$ (for some X, A, and B), add a child node labeled $X \cup \{ A, B \}$
  - If a leaf of the tree is labeled with $X \cup \{ A \vee B \}$ (for some X, A, and B), add two child node labeled $X \cup \{ A \}$ and $X \cup \{ B \}$
  - If a leaf of the tree is labeled with $X \cup \{ P, \neg P \}$ or $X \cup \{ \bot \}$ (for some X and propositional symbol P), mark the leaf as *contradictory*
- Note that a propositional tableau is always finite

Propositional Tableaux (II)

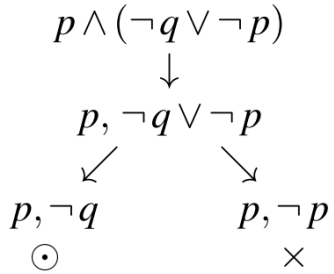
- Note that
  - The nodes of a tableau are labeled with sets of propositions in negation form
  - Two rules are used to expand a tableau by adding children to leafs ($\wedge$-rule and $\vee$-rule)
  - One rule is used to identify contradictory leafs (¬-rule)
  - A tableau is *complete* if no rules are further applicable
  - The leafs (contradictory or not) of a complete tableau are labeled with sets of literals because expansion rules remove connectives from propositions

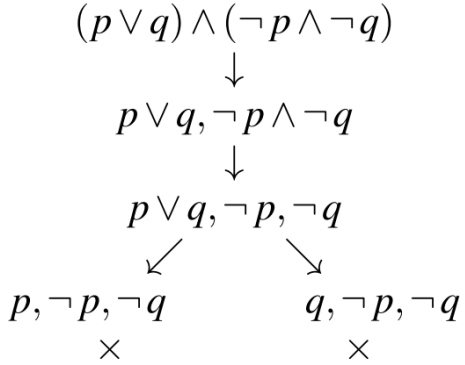
Propositional Tableaux (III)

- Propositional tableaux are used to check satisfiability
  - A path from the root of a complete tableau to a leaf is *closed* if the leaf is contradictory, otherwise it is *open*
  - A tableau is *closed* if all its paths are closed
  - The set of propositions that labels the root of a complete tableau is *unsatisfiable* if and only if the tableau is closed
  - The label of the leaf of an open path identifies a set of models of the set of propositions that labels the root
    - For example, if an open path ends with the label { p, ¬q }, then any interpretation *I* such that *I*(*p*)=T and *I*(*q*)=F is a model of the set of propositions that labels the root of the tableau

Propositional Tableaux (IV)

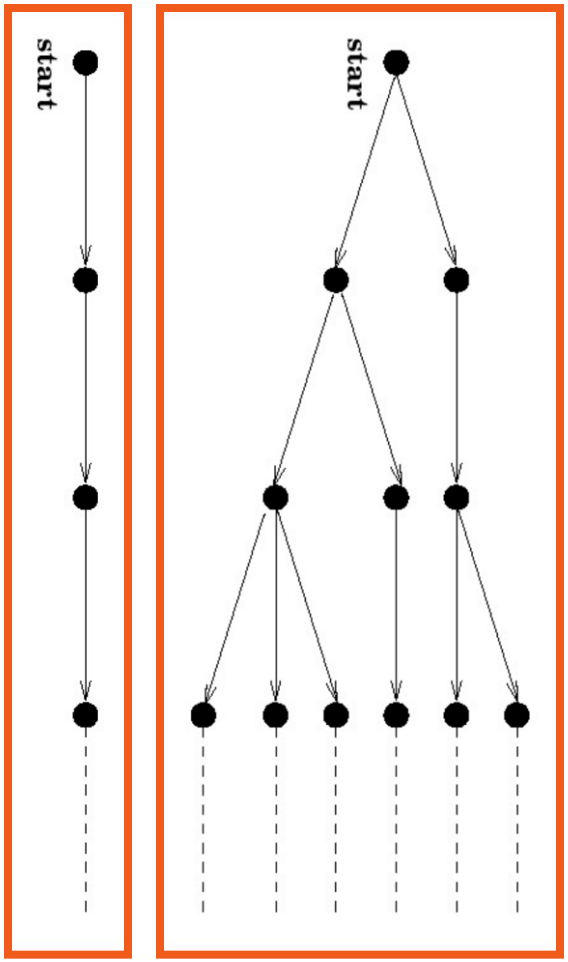
- Therefore
  - A set of propositions S is unsatisfiable if and only if all the leafs of a tableau are marked as contradictory
  - Given a set of propositions S and a proposition A, to prove that $S \models A$, it is sufficient to consider $S' = S \cup \{\neg A\}$ and prove, by building a tableau, that S' is unsatisfiable





Temporal Logics (I)

- In classical logics, propositions are interpreted *statically*
  - Interpretations statically assign truth values to propositions
  - Interpretations are not allowed to change over time
- Software systems are characterized by states that change over time
  - Classical logics are not sufficient



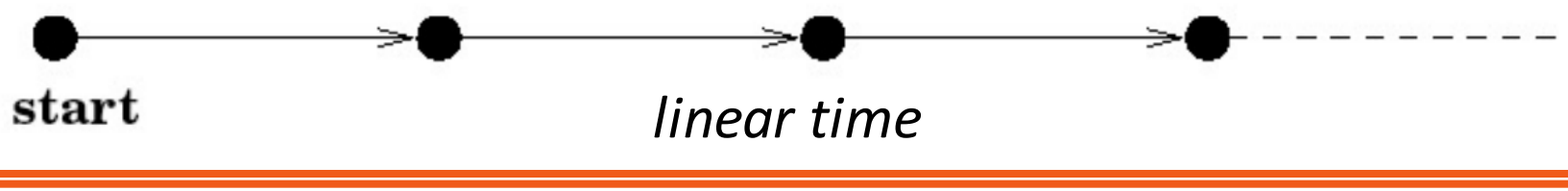
Temporal Logics (II)

- In classical logic, propositions are interpreted in the scope of *a single static world*
  - A propositional symbol *p* (e.g., *It is Sunday)* must be either true or false now and forever
  - Propositions are assembled using connectives, and the truth values of propositions are determined statically from the truth values of propositional symbols
- In temporal logics, the interpretation takes place in the scope of *a set of worlds*
  - Thus, *p* (e.g., *It is Sunday)* may be interpreted as T in some worlds, but not in others

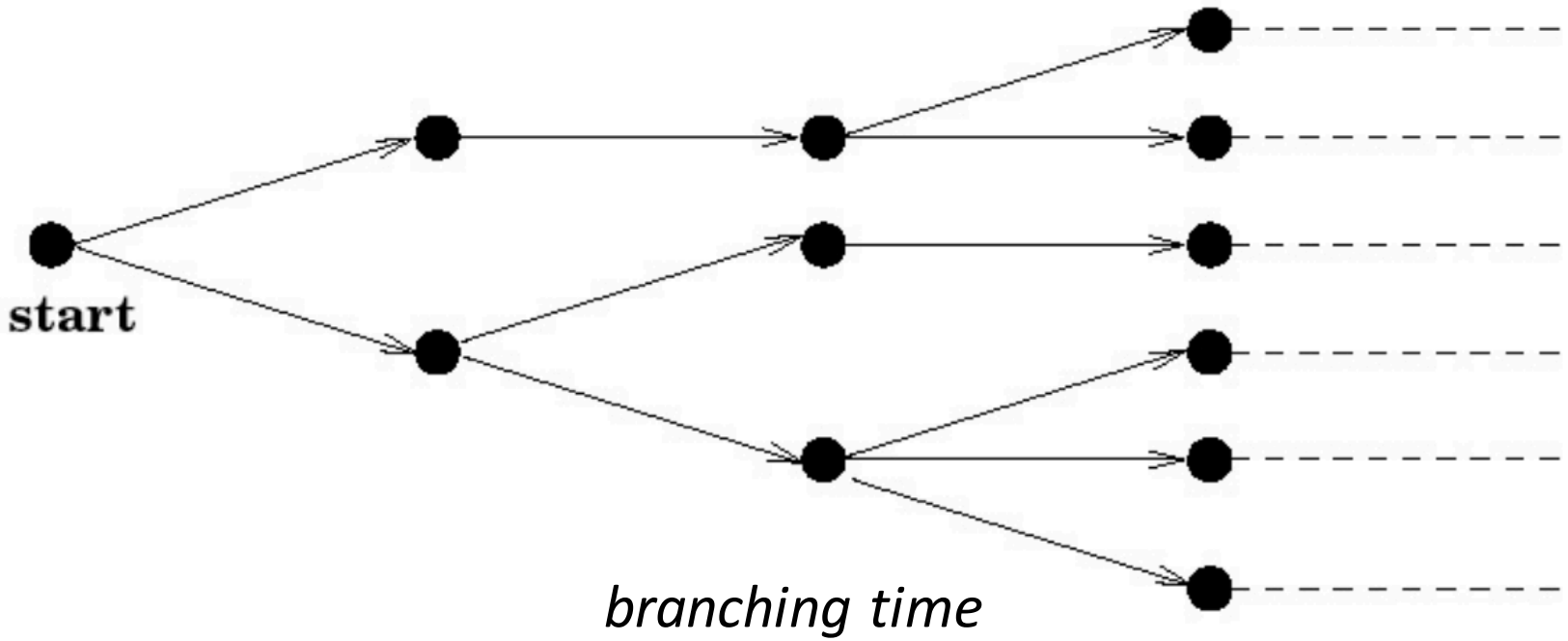
Temporal Logics (III)

- The set of worlds that characterizes temporal logics correspond to *moments in time*
- The particular model of time for a given temporal logic is captured by a *(temporal) accessibility relation* among worlds
  - Different temporal logics may have different accessibility relations that link worlds
- Temporal logics extend propositional logic with *temporal (modal) operators* to navigate worlds using the accessibility relation of the chosen logic

Two Models of Time



linear time



branching time

Basics of Linear Temporal Logic (I)

- The simple Linear Temporal Logic (LTL) has an accessibility relation that describes a discrete, linear model of time isomorphic to the natural numbers
- For example,

  $G((\neg p \vee \neg t) \rightarrow X\neg b)$

  where
  - *p* is *have passport*
  - *t* is *have boarding pass*
  - *b* is *board the plane*

LTL Operators	
X*p*	*p* is true in the *neXt* moment in time
G*p*	*p* is *Globally* true in all future moments in time
F*p*	*p* is true in some *Future* moment in time
*p*U*q*	*p* is true *Until q* is true

Basics of Linear Temporal Logic (III)

- Given a software system that ensures that
  - G(*requested*→F*received*)
  - G(*received*→X*processed*)
  - G(*processed*→FG*done*)

LTL Operators	
X*p*	*p* is true in the *neXt* moment in time
G*p*	*p* is *Globally* true in all future moments in time
F*p*	*p* is true in some *Future* moment in time
*p*U*q*	*p* is true *Until q* is true

- From the above LTL propositions, we want to be able to deduce that the following LTL proposition is false

$$G\mathit{requested} \wedge G\neg\mathit{done}$$

Syntax (I)

- Given a countable set of propositional symbols $P \neq \emptyset$, the set *LTL*[*P*] of *LTL propositions* over *P* is constructed as follows
  - Every propositional symbol is an LTL proposition
  - $\top$ (*top*) and $\bot$ (*bottom*) are LTL propositions (it is always assumed that $\top \notin P$ and $\bot \notin P$)
  - If *A* is an LTL proposition, then $\neg A$ is an LTL proposition
  - If *A* and *B* are LTL propositions, then $A \wedge B$, $A \vee B$, $A \rightarrow B$, $A \equiv B$ are LTL proposition
  - *If A and B are LTL propositions, then* X*A,* G*A,* F*A, A*U*B are LTL proposition*
  - Nothing else is an LTL proposition
- The syntax of LTL is a simple extension of the syntax of propositional logic

Syntax (II)

- Syntactic conventions
  - Parentheses can be used to group propositions
    - Outer parentheses can be omitted
    - Square bracket and curly brace are also usable, but only parentheses are normally used
  - Binary connectives are *left associative*
    - Including U
  - The precedence of connectives and operators follows the order ¬, G, F, X, U, $\wedge$, $\vee$, $\rightarrow$, $\equiv$

Semantics (I)

- The interpretation of LTL propositions is based on the following assumptions
  - Each moment in time is represented as a natural number (discrete, linear time)
  - For each moment in time, a *world* is represented as a set of LTL propositions that are true
- The *interpretation function*

$$I : P \times \mathbb{N} \rightarrow \boldsymbol{B}$$

where $\boldsymbol{B}$ = { F, T }, maps each propositional symbol to a truth value in $\boldsymbol{B}$ for each moment in time

Semantics (II)

- Given an interpretation I over P, the interpretation

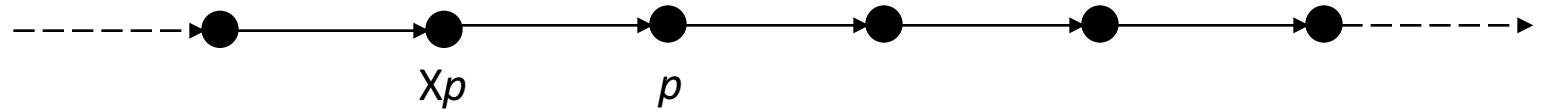
$$G_I : LTL[\mathrm{P}]\times\mathbb{N} \rightarrow \boldsymbol{B}$$

of an arbitrary LTL proposition in $LTL[P]$ can be computed as follows

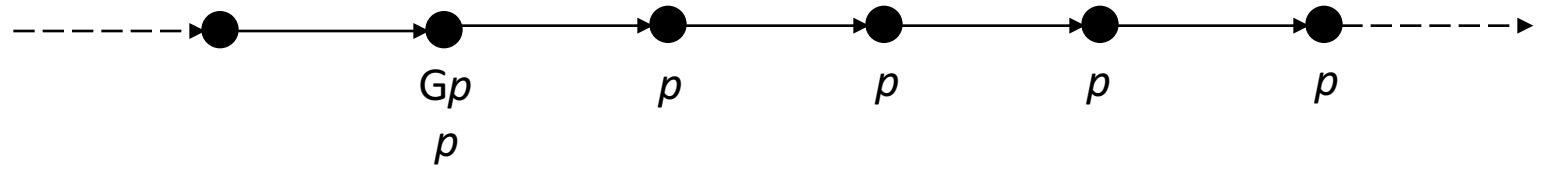
  - $G_I(A,i)=I(A,i)$ if $A\in P$
  - $G_I(\top,i)=\mathrm{T}$ and $G_I(\bot,i)=\mathrm{F}$
  - $G_I(\neg A,i)=\mathrm{F}$ if $G_I(A,i)=\mathrm{T}$, otherwise $G_I(\neg A,i)=\mathrm{T}$
  - $G_I(A \wedge B,i)=\mathrm{T}$ if $G_I(A,i)=\mathrm{T}$ and $G_I(B,i)=\mathrm{T}$, otherwise $G_I(A \wedge B,i)=\mathrm{F}$
  - $G_I(A \vee B,i)=\mathrm{F}$ if $G_I(A,i)=\mathrm{F}$ and $G_I(B,i)=\mathrm{F}$, otherwise $G_I(A \vee B,i)=\mathrm{T}$
  - $G_I(A \rightarrow B,i)=\mathrm{T}$ if and only if $G_I(A,i)=\mathrm{F}$ or $G_I(B,i)=\mathrm{T}$
  - $G_I(A \equiv B,i)=\mathrm{T}$ if and only if $G_I(A,i)=G_I(B,i)=\mathrm{T}$ or $G_I(A,i)=G_I(B,i)=\mathrm{F}$
  - *…continues in next slides with the semantics of temporal operators*

Semantics of Temporal Operators (I)

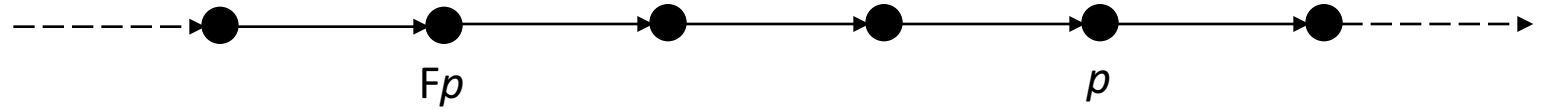
- X*p* is used to state that *p* is true in the next moment in time



- G*p* is used to state that *p* is true in all future moments



- F*p* is used to state that *p* is true in some future moment



- *p*U*q* is used to state that *p* is true until *q* becomes true

Semantics of Temporal Operators (II)

- Given an interpretation I over P, the interpretation

$$G_I : LTL[\mathrm{P}]\times\mathbb{N} \rightarrow \boldsymbol{B}$$

of an arbitrary LTL proposition in $LTL[P]$ can be computed as follows

  - *…continued from the semantics of connectives in previous slides*
  - $G_I(\mathrm{X}A,i)$=T if and only if $G_I(A,i+1)$=T
  - $G_I(\mathrm{F}A,i)$=T if and only if there exists $j\in\mathbb{N}$ such that $j\geq i$ and $G_I(A,j)$=T
  - $G_I(\mathrm{G}A,i)$=T if and only if for all $j\in\mathbb{N}$, if j≥i then $G_I(A,j)$=T
  - $G_I(A\mathrm{U}B,i)$=T if and only if there exists $j\in\mathbb{N}$ such that $j\geq i$ and $G_I(\mathrm{B},j)$=T and for all $k\in\mathbb{N}$, if $i\leq k<j$ then $G_I(A,k)$=T

Satisfiability and Tautologies

- Given an interpretation M, a moment in time $i \in \mathbb{N}$, and an LTL proposition A
  - $\langle M,i \rangle \models A$ *(M satisfies A at i)* if and only if $G_M(A,i)$=T
  - $\langle M,i \rangle \not\models A$ (*M does not satisfy A at i*) if and only if $G_M(A,i)$=F
- An interpretation M is a *model* for a proposition A if and only if there exists some $i \in \mathbb{N}$ such that $\langle M,i \rangle \models A$
- An LTL proposition A is
  - *Satisfiable* if and only if there exists a model for A
  - A *tautology* ($\models A$) if and only if for any interpretation M and any moment in time $i \in \mathbb{N}$, $\langle M,i \rangle \models A$

Models (I)

- Given an interpretation M defined over a set of propositional symbols P, the following rules can be used to *check* if $\langle M,i\rangle \models A$ (M satisfies A at $i \in \mathbb{N}$)
  - $\langle M,i\rangle \models p$ if and only if $p \in P$ and $M(p,i) = T$
  - $\langle M,i\rangle \models \top$ and $\langle M,i\rangle \not\models \bot$ for all $i \in \mathbb{N}$
  - $\langle M,i\rangle \models \neg A$ if and only if $\langle M,i\rangle \not\models A$
  - $\langle M,i\rangle \models A \wedge B$ if and only if $\langle M,i\rangle \models A$ and $\langle M,i\rangle \models B$
  - $\langle M,i\rangle \models A \vee B$ if and only if $\langle M,i\rangle \models A$ or $\langle M,i\rangle \models B$
  - $\langle M,i\rangle \models A \rightarrow B$ if and only if $\langle M,i\rangle \not\models A$ or $\langle M,i\rangle \models B$
  - $\langle M,i\rangle \models A \equiv B$ if and only if $\langle M,i\rangle \models A$ and $\langle M,i\rangle \models B$, or $\langle M,i\rangle \not\models A$ and $\langle M,i\rangle \not\models B$
  - *…continues to model checking of temporal operators in next slide*

Models (II)

- Given an interpretation M defined over a set of propositional symbols P, the following rules can be used to *check* if $\langle M,i\rangle \models A$ (M satisfies A at $i \in \mathbb{N}$)
  - *…continued from model checking of connectives in previous slide*
  - $\langle M,i\rangle \models XA$ if and only if $\langle M,i+1\rangle \models A$
  - $\langle M,i\rangle \models FA$ if and only if there exists $j \in \mathbb{N}$ such that $j \geq i$ and $\langle M,j\rangle \models A$
  - $\langle M,i\rangle \models GA$ if and only if for all $j \in \mathbb{N}$, if $j \geq i$ then $\langle M,j\rangle \models A$
  - $\langle M,i\rangle \models AUB$ if and only if there exists $j \in \mathbb{N}$ such that $j \geq i$, $\langle M,j\rangle \models B$, and for all $k \in \mathbb{N}$, if $i \leq k < j$ then $\langle M,k\rangle \models A$

Logical Equivalences (I)

- LTL propositions A and B are logically equivalent ($A \leftrightarrow B$) if and only if $\models (A \equiv B)$
  - For any interpretation M and any moment in time $i \in \mathbb{N}$, $\langle M,i \rangle \models A$ if and only if $\langle M,i \rangle \models B$
- The following are some relevant logical equivalences
  - Inner propagation of negations
    - $\neg \mathrm{X}A \leftrightarrow \mathrm{X}\neg A$, $\neg \mathrm{F}A \leftrightarrow \mathrm{G}\neg A$, $\neg \mathrm{G}A \leftrightarrow \mathrm{F}\neg A$
  - F (and G) can be written in terms of U
    - $\mathrm{F}A \leftrightarrow \top \mathrm{U}A$
  - Simplifications
    - $\mathrm{FF}A \leftrightarrow \mathrm{F}A$, $\mathrm{GG}A \leftrightarrow \mathrm{G}A$, $A\mathrm{U}(A\mathrm{U}B) \leftrightarrow A\mathrm{U}B$

Logical Equivalences (II)

- The following are other relevant logical equivalences
  - Distributivity
    - $X(A \vee B) \leftrightarrow XA \vee XB$, $X(A \wedge B) \leftrightarrow XA \wedge XB$, $X(AUB) \leftrightarrow (XA)U(XB)$
      $F(A \vee B) \leftrightarrow FA \vee FB$, $G(A \wedge B) \leftrightarrow GA \wedge GB$
      $AU(B \vee C) \leftrightarrow AUB \vee AUC$, $AU(B \wedge C) \leftrightarrow AUB \wedge AUC$
  - Explicitation
    - $B \vee (A \wedge X(AUB)) \leftrightarrow AUB$, $A \wedge X(GA) \leftrightarrow GA$, $A \vee X(FA) \leftrightarrow FA$
  - The following equivalence is useful to turn LTL propositions into negation form
    - $\neg(AUB) \leftrightarrow (\neg BU(\neg A \wedge \neg B)) \vee G\neg B$

LTL Tableaux (I)

- LTL tableaux are *directed graphs* (not trees) used to check the satisfiability of sets of LTL propositions
  - Note, an LTL tableau is a *labeled* graph, and therefore is contains no nodes with the same label
  - A new node is not added if its label already labels another node, instead the existing node is linked
- The LTL propositions in the set to be analyzed are turned into negation form using the following logical equivalences
  - $A \equiv B \leftrightarrow (A \rightarrow B) \wedge (B \rightarrow A)$
  - $A \rightarrow B \leftrightarrow \neg A \vee B$
  - $\neg\neg A \leftrightarrow A$, $\neg(A \vee B) \leftrightarrow (\neg A \wedge \neg B)$, $\neg(A \wedge B) \leftrightarrow (\neg A \vee \neg B)$
  - $\neg \mathrm{X} A \leftrightarrow \mathrm{X} \neg A$, $\neg \mathrm{F} A \leftrightarrow \mathrm{G} \neg A$, $\neg \mathrm{G} A \leftrightarrow \mathrm{F} \neg A$, $\neg(A \mathrm{U} B) \leftrightarrow (\neg B \mathrm{U} (\neg A \wedge \neg B)) \vee \mathrm{G} \neg B$

LTL Tableaux (II)

- The following *propositional rules* are applied starting from the root of the tableau labeled with the set of LTL formulae in negation form
  - If a leaf of the tableaux is labeled with $S \cup \{ P, \neg P \}$ or $S \cup \{ \bot \}$ (for some S and propositional symbol P), stop expanding the leaf and mark it as contradictory
  - If a leaf of the tableaux is labeled with $S \cup \{ A \bigwedge B \}$ (for some S, A, and B), add a child node labeled $S \cup \{ A, B \}$
  - If a leaf of the tableaux is labeled with $S \cup \{ A \bigvee B \}$ (for some S, A, and B), add two child nodes labeled $S \cup \{ A \}$ and $S \cup \{ B \}$

LTL Tableaux (III)

- The following *temporal rules* are applied starting from the root of the tableau labeled with the set of LTL formulae in negation form
  - If a leaf of the tableaux is labeled with $S \cup \{ G A \}$ (for some S and A), add one child node labeled $S \cup \{ A, XGA \}$
  - If a leaf of the tableaux is labeled with $S \cup \{ FA \}$ (for some S and A), add two children labeled $S \cup \{ A \}$ and $S \cup \{ XFA \}$
  - If a leaf of the tableaux is labeled with $S \cup \{ AUB \}$ (for some S, A, and B), add two children labeled $S \cup \{ B \}$ and $S \cup \{ A, X(AUB) \}$

LTL Tableaux (IV)

- The following *loop rule* is applied if no other rule is applicable starting from the root of the tableau labeled with the set of LTL formulae in negation form
  - If a leaf node is labeled only with literals and LTL propositions structured as X*P*, for some *P*, then
    - If the set *S'* of the propositions that are captured by the X operator in the label of the node is not a subset of any label in the tableau (*loop checking*), add one child to the leaf node labeled with *S'*
    - Otherwise, connect the leaf node with the node whose label is a superset of *S'*
- Note that loop checking is sufficient to ensure the termination of the construction of all tableaux

LTL Tableaux (V)

Classical Rules	
$\dfrac{A \wedge B, S}{A, B, S}\ (\wedge)$	$\dfrac{A \vee B, S}{A, S \qquad B, S}\ (\vee)$
Temporal and Loop Rules	
$\dfrac{\Box A, S}{A, \bigcirc \Box A, S}\ (\Box)$	$\dfrac{\Diamond A, S}{A, S \qquad \bigcirc \Diamond A, S}\ (\Diamond)$
$\dfrac{A \mathcal{U} B, S}{B, S \qquad A, \bigcirc (A \mathcal{U} B), S}\ (\mathcal{U})$	
$\dfrac{\Lambda, \bigcirc A_1, ..., \bigcirc A_n}{A_1, ..., A_n}\ (\bigcirc)$ Λ is a set of literals	

Legend

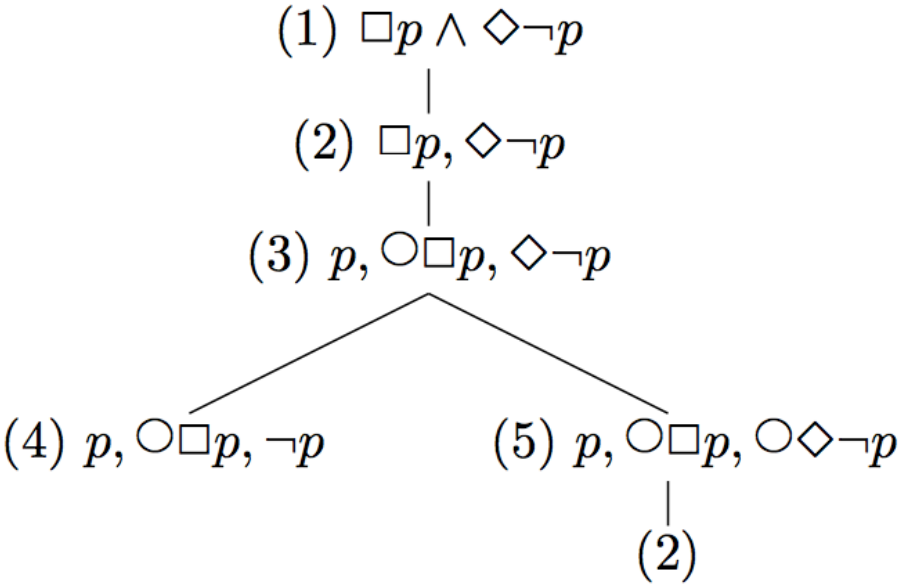
□=G

◇=F

◯=X

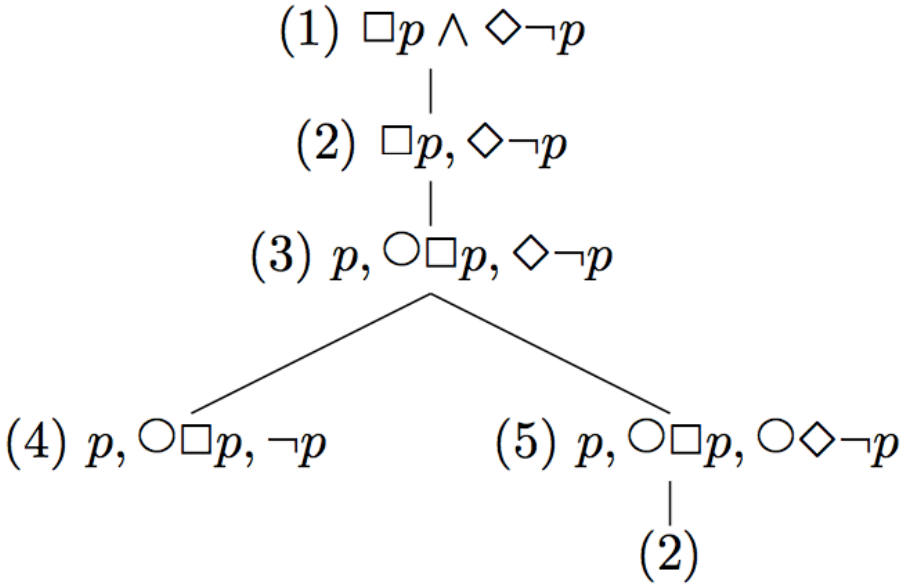
LTL Satisfiability (I)

- *LTL (semantic) tableaux* are directed graphs used to reason on sets of LTL propositions
  - The nodes of the tableau are labeled with sets of formulae in negation form
  - The loop rule is applied if no other rule is applicable to step in time
  - A tableau is *complete* if no rules are applicable
- Note: □=G, ◇=F, ◯=X



LTL Satisfiability (II)

- The definition of *closed tableau* is more involved than in the propositional case
- An *eventuality* is an LTL proposition structured as F*E* or *A*U*E*
- An eventuality F*E* or *A*U*E* is *satisfied* in a node *n* if there exists a path starting from *n* that includes a node whose label contains *E*



LTL Satisfiability (III)

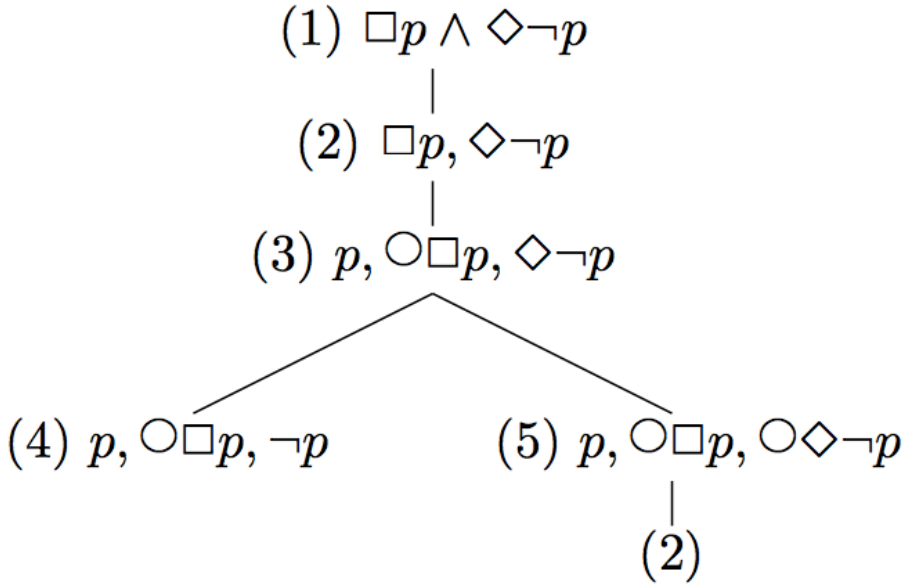
- Given a complete tableau, a node can be *cancelled* if
  - The node is contradictory
  - The label of the node contains an eventuality that is *not satisfied* in the node
  - All children of the node are marked as *cancelled*
- A complete tableau is *closed* if and only if its root can be cancelled
- The set of LTL propositions that labels the root of a complete tableau is unsatisfiable if and only if the tableau is closed

LTL Satisfiability (IV)

- Given a complete tableau, if it is not closed it is *open*
- The paths starting from the root in an open tableau provide information on the models of the set of LTL propositions that label the root of the tableau
  - The explicit construction of such models requires extending the labels of the nodes with additional information
- The explicit construction of models from open tableaux is not relevant here

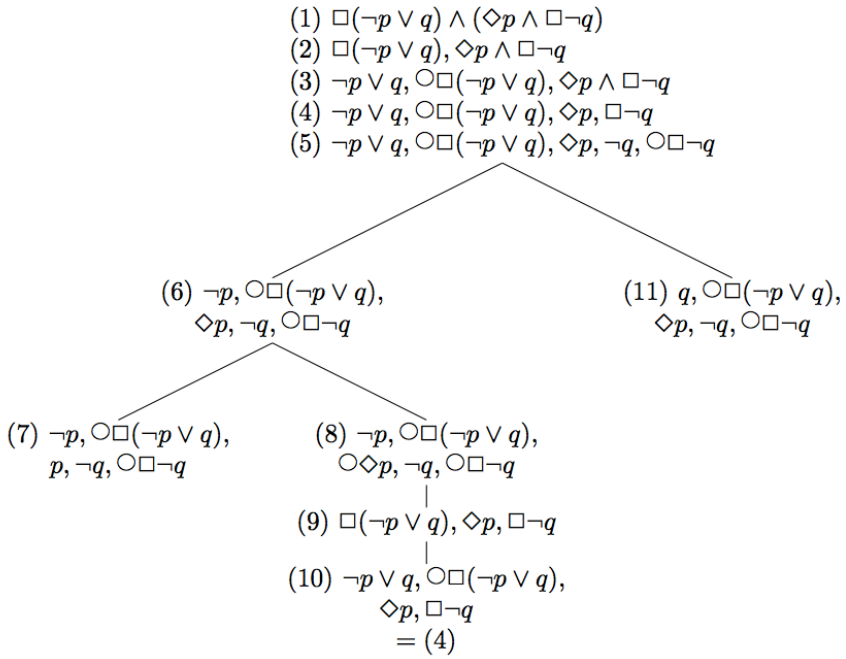
LTL Satisfiability – Example (I)

- The tableau in the example is closed because
  - Node (4) can be cancelled because it is contradictory
  - Node (2) can be cancelled because the eventuality F¬*p* is not satisfied in the node
  - Node (1) can be cancelled because its children are all marked as cancelled



LTL Satisfiability – Example (II)

- The tableau in the example is closed because
  - Node (7) and (11) can be cancelled because they are contradictory
  - Node (4) can be cancelled because the eventuality F*p* is not satisfied in the node
  - Node (1) can be cancelled because its children are all marked as cancelled

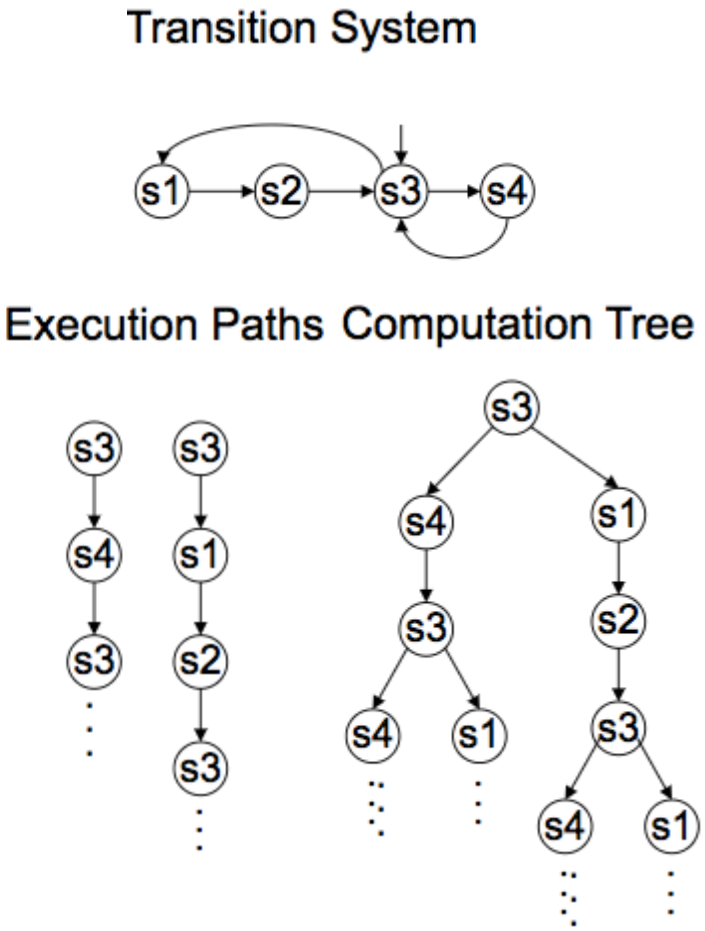


Temporal Logics and Software Engineering

- Temporal logics were originally developed to reason on verb tenses in natural language

- In software engineering, they achieved a significant role in the *formal specification and verification* of concurrent systems

- In 1996, *A. Pnueli* received the Turing Award for his *seminal work introducing temporal logic into Computing Science and for outstanding contributions to program and systems verification*



Asynchronous Concurrent Reactive Systems

- Temporal logics are useful to reason on asynchronous concurrent reactive systems
  - *Reactive systems* are systems that interact with their environment and are usually not supposed to terminate
    - They are often called agents
    - E.g., communicating software systems or hardware devices
  - *Concurrent systems* consist of a set of parts that execute concurrently
    - In *asynchronous (or interleaved) systems* only one component takes a step at a time
    - In *synchronous systems* all components take steps at the same time

Modeling Systems

- The *modeling* of a system is intended to construct a (*possibly formal*) specification of the system to *abstract* away irrelevant details
- The specification is described in terms of
  - The state of the system, which is a snapshot of the values of the parameters that characterize the system
  - The transitions of the system, which describe how the state of the system changes over time because of events and actions
  - The computation of the system, which is the (possibly infinite) sequence of states activated by transitions

Kripke Structures (I)

- Kripke structures are transition diagrams (sometimes called *transition systems*) that describe the dynamic behavior of a reactive system
- Kripke structures consist of
  - A nonempty set of *states*
  - A nonempty set of *transitions* among states
  - A nonempty set of *propositions* to label states
- A *path* in a Kripke structure represents a possible computation of the described system

Kripke Structures (II)

- Formally, a Kripke structures is a 5-tuple $K=<S,I,R,P,L>$, where
  - S is a nonempty set of *states*
  - $I \subseteq S$ is a nonempty set of *initial states*
  - $R \subseteq S \times S$ is an accessibility relation, a nonempty set of *transitions* such that R is *left-total*

$$\forall s \in S,\ \exists s' \in S : (s,s') \in R$$

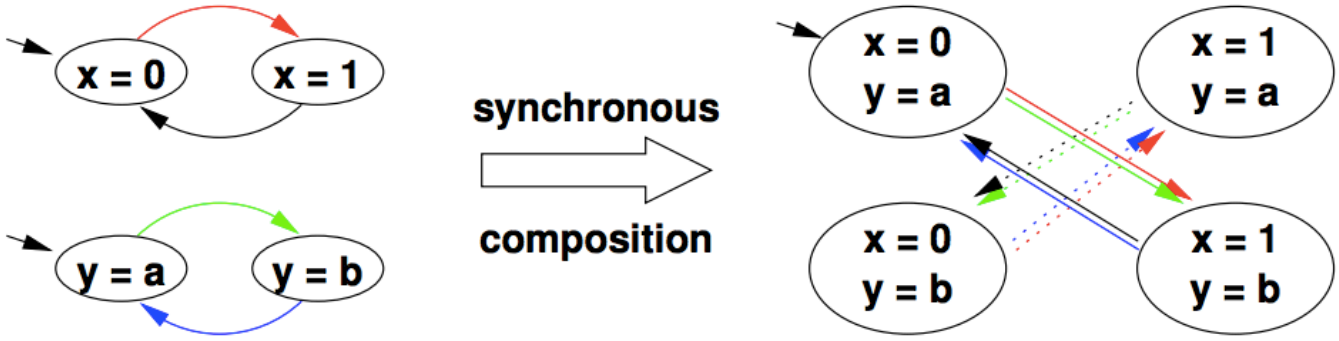
  - P is a countable set of propositional symbols used to construct $Prop[P]$
  - $L : S \rightarrow 2^{Prop[P]}$ is a *labeling function*
- A path π in a Kripke structure K from a state $s_0 \in S$ is an *infinite* sequence of states

$$\pi = s_0\, s_1\, s_2 \ldots$$

such that $(s_i, s_{i+1}) \in R$ for all $i \in \mathbb{N}$ (because R is left-total)

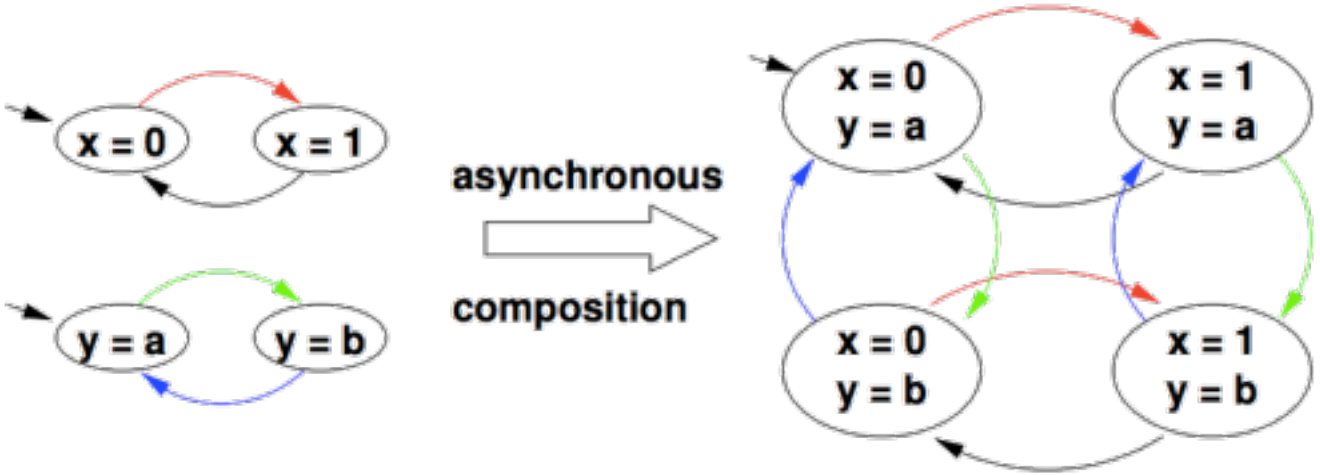
Complex Kripke Structures (I)

- *Complex Kripke structures* are typically obtained by composition of simple Kripke structures
- Kripke structures can be combined using *synchronous composition*
  - The states of components evolve synchronously
  - At each moment in time, every component performs a transition



Complex Kripke Structures (II)

- *Complex Kripke structures* are typically obtained by composition of simple Kripke structures
- Kripke structures can be combined using a*synchronous composition*
  - The changes of the states of components are *interleaved*
  - At each moment in time, only one component performs a transition



Process Languages (I)

- Typically, Kripke structures are described using high level languages called *process (meta) languages*
- Process languages allow
  - Describing parts (normally called processes)
  - Aggregating parts using synchronous and/or asynchronous composition
- The description of a part normally contains
  - A set of variables describing the state of the part
  - Initial values for variables
  - Procedures to describe transitions

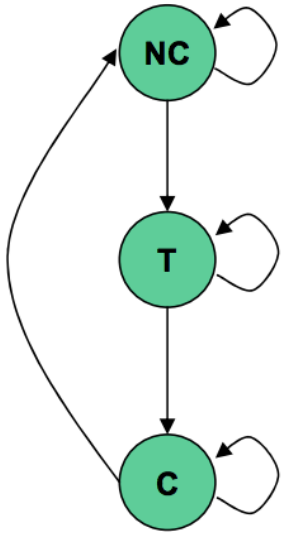
Process Languages (II)

```
mtype = { NONCRITICAL, TRYING, CRITICAL };
mtype state[2];

proctype process(int id) {
beginning:
noncritical:
        state[id] = NONCRITICAL;
        if ::goto noncritical; ::true; fi;
trying:
        state[id] = TRYING;
        if ::goto trying; ::true; fi;
critical:
        state[id] = CRITICAL;
        if ::goto critical; ::true; fi;
        goto beginning;
}

init { run process(0); run process(1); }
```

Description of a Kripke structure in *Promela* (*Process Meta Language*) for *SPIN* (*Simple Promela Interpreter*)



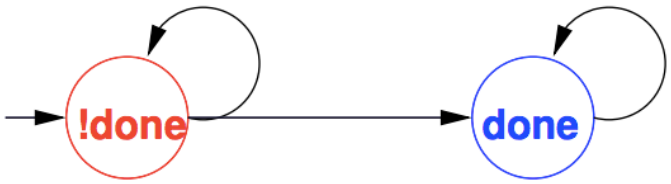
Computation Tree Logic (I)

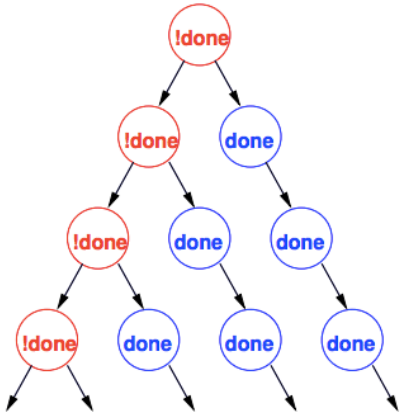
- Computation Tree Logic (CTL) is a temporal logic that overcomes known limitations of LTL
  - CTL is a *branching time* logic, while LTL is a linear time logic
  - CTL introduces *path quantifiers*, which are not allowed in LTL because of linear time

- The syntax of CTL is constructed starting from the syntax of propositional logic by adding the following
  - Existential temporal operators: EG, EF, EX, E/U
  - Universal temporal operators: AG, AF, AX, A/U

Computation Tree Logic (II)

- Informally, the semantics of CTL is described using Kripke structures *linearized as trees* in which states are interpreted as *possible worlds (or states)*
  - The root of the tree is the *current world*
  - The accessibility relation of the Kripke structure allows navigating possible worlds





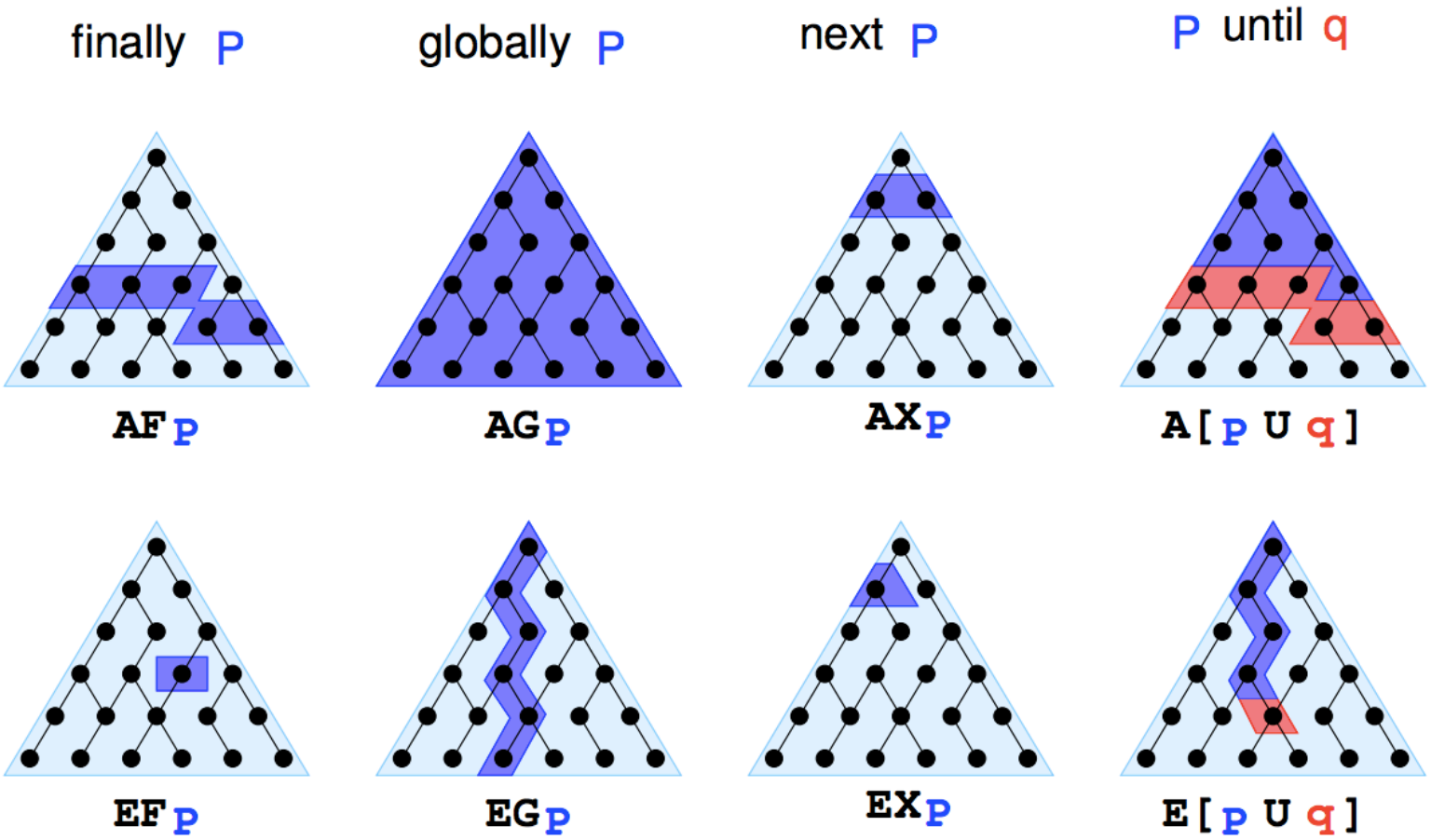
Computation Tree Logic (III)

- Informally, given two CTL propositions *P* and *Q*, the following are CTL propositions that use existential temporal operators
  - EG*P*, to state that there exists a path starting from the current world in which *P* is always true
  - EF*P*, to state that there exists a path starting from the current world in which *P* will eventually become true
  - EX*P*, to state that there exists a path starting from the current world in which *P* is true in the next world along the path
  - E*P*U*Q*, to state that there exists a path starting from the current world in which *P* is true until *Q* would eventually become true
- The formal semantics of existential temporal operators is obtained by adapting to at least one path the corresponding semantics of LTL temporal operators

Computation Tree Logic (IV)

- Informally, given two CTL propositions *P* and *Q*, the following are CTL propositions that use universal temporal operators
  - AG*P*, to state that in all paths starting from the current world, *P* is always true
  - AF*P*, to state that in all paths starting from the current world, *P* will eventually become true
  - AX*P*, to state that in all paths starting from the current world, *P* is true in the next world along the path
  - A*P*U*Q*, to state that in all paths starting from the current world, *P* is true until *Q* would eventually become true
- The formal semantics of universal temporal operators is obtained by adapting to all paths the corresponding semantics of LTL temporal operators

Computation Tree Logic (V)



Computation Tree Logic (VI)

- Note that all CTL temporal operators can be expressed in terms of EG, EX, and E/U
  - $EF P \leftrightarrow E\top U P$
  - $AX P \leftrightarrow \neg EX \neg P$
  - $AF P \leftrightarrow \neg EG \neg P$
  - $AG P \leftrightarrow \neg EF \neg P$
  - $A P U Q \leftrightarrow \neg EG \neg Q \bigwedge \neg E \neg Q U (\neg P \bigwedge \neg Q)$

CTL vs LTL

- Several CTL propositions that contain path quantifiers cannot be expressed in LTL
  - For example, AG($P \rightarrow$ EFQ)
- Several LTL propositions that select a range of paths using a property cannot be expressed in CTL
  - For example, GF$P \rightarrow$ GFQ
- Several LTL propositions can be expressed in CTL, and vice versa
  - For example, FP, G$\neg(P \wedge Q)$, G($P \rightarrow$ FQ), GFP
- CTL and LTL have *incomparable* expressive power
  - The choice between LTL and CTL depends on the use and on the personal preferences

CTL* (I)

- Computation Tree Logic* (CTL*) is a temporal logic that combines the expressive power of LTL and CTL
- The syntax for CTL* allows two types of propositions
  - *State propositions*
  - *Path propositions*
- If S is a propositional symbol, P and Q are state propositions, and W is a path proposition, then the following are *state propositions*

$$S, \top, \bot, \neg P, P \wedge Q, P \vee Q, \mathrm{A}W, \mathrm{E}W$$

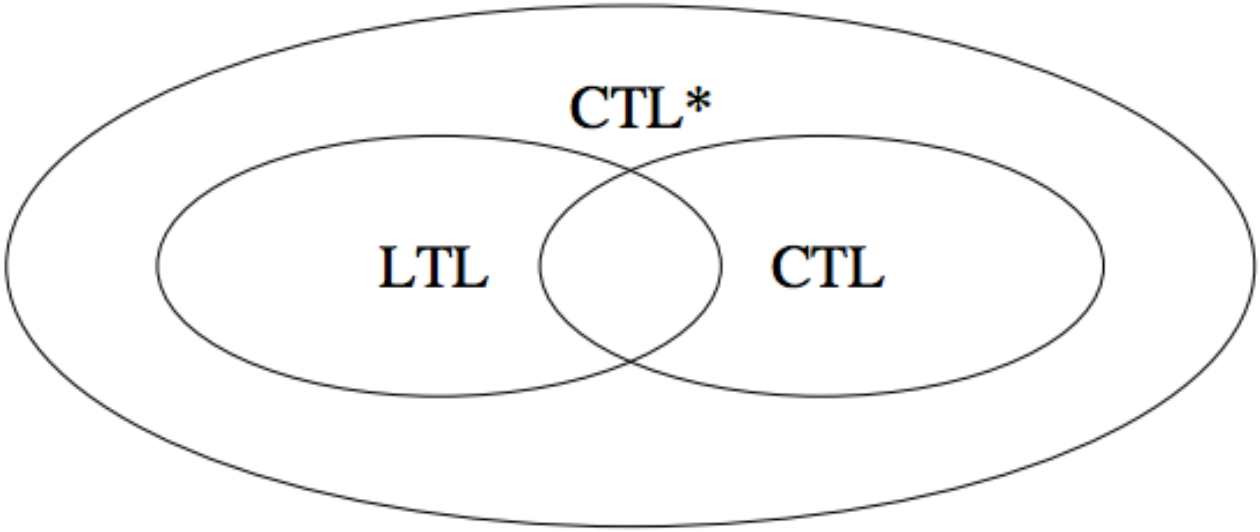
- If V and W are path propositions and P is a state proposition, then the following are *path propositions*

$$P, \neg W, V \wedge W, V \vee W, \mathrm{X}W, \mathrm{G}W, \mathrm{F}W, V\mathrm{U}W$$

CTL* (II)

- CTL* subsumes both CTL and LTL
- Given a set of propositional symbols *P*, then

$$LTL[P] \cup CTL[P] \subset CTL^*[P]$$



Temporal Logics for Asynchronous Concurrent Reactive Systems

- Relevant properties in illustrative asynchronous concurrent reactive systems
  - No more than one processor (in a 2-processor system) shall have a cache line in write mode
    $P = \{p1, p2\}$, with *pi* meaning *processor i has the cache line in write mode*

    $$G\neg(p1 \wedge p2)$$
  - The grant signal must be asserted some time after the request signal is asserted
    $P = \{r, g\}$, with *r* meaning *request signal is asserted* and *g* meaning *grant signal is asserted*

    $$G(r \rightarrow Fg)$$
  - A request must receive an acknowledgement, and the request should stay asserted until the acknowledgment is received
    $P = \{r, a\}$, with *r* for *request* and *a acknowledgment*

    $$G(r \rightarrow (r \, U \, a))$$

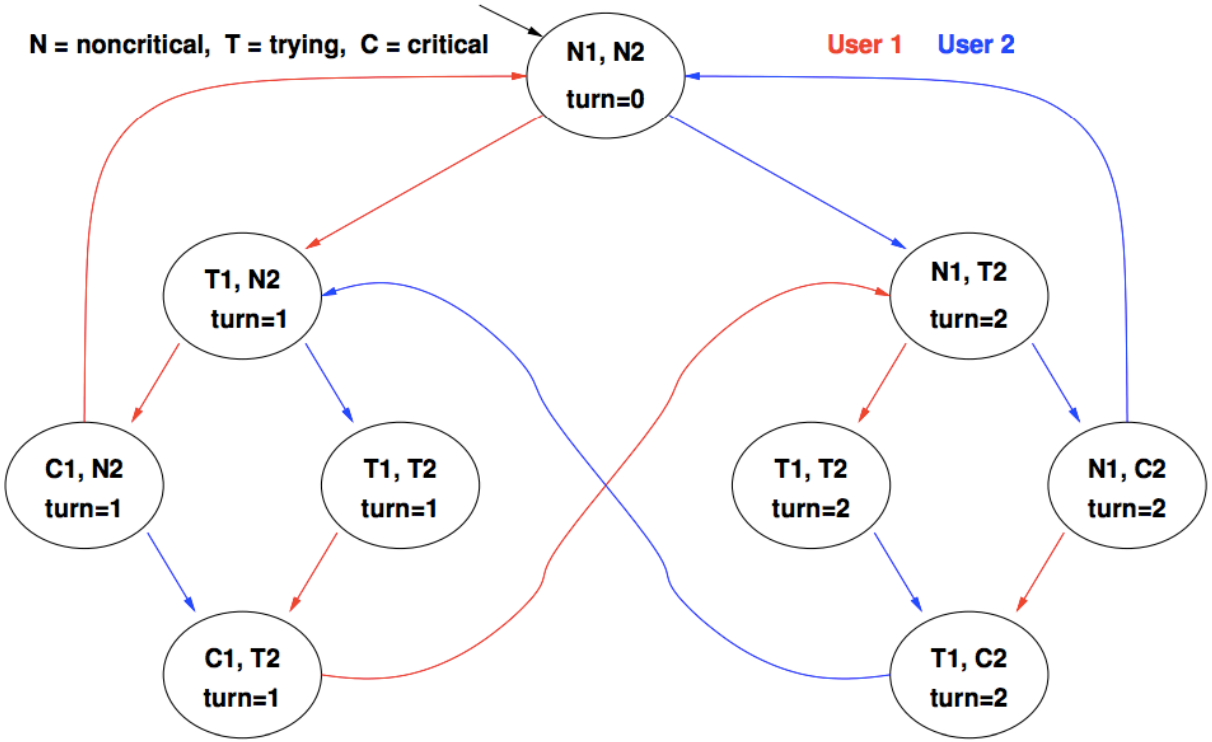
Safety, Liveness, Fairness (I)

- Much of the popularity of temporal logics was achieved because several concepts related to concurrent software systems can be formally expressed and studied
- Normally, the properties of concurrent software systems that are studied using temporal logics are grouped into three categories
  - *Safety* properties
  - *Liveness* properties
  - *Fairness* properties

Safety, Liveness, Fairness (II)

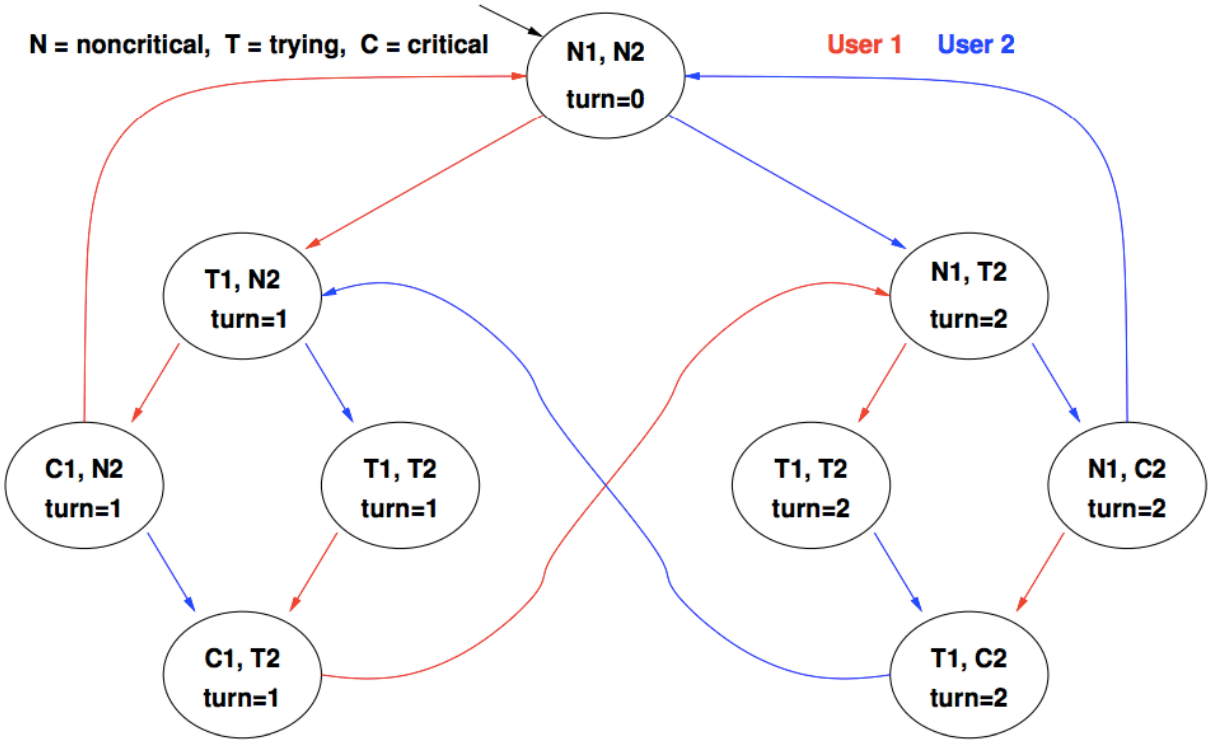
- Safety properties are requested to ensure that *nothing bad will ever happen*
  - G¬(*temperature* > 100)
- Liveness properties are requested to ensure that *something good will eventually happen*
  - G(*started*→F*terminated*)
- (Strong) Fairness properties are requested to ensure that *if something is requested infinitely often, then it will be served infinitely often*
  - GF*ready*→GF*execute*

Mutual Exclusion



The *turn* variable is used to track the first *user* that tries to work in mutual exclusion

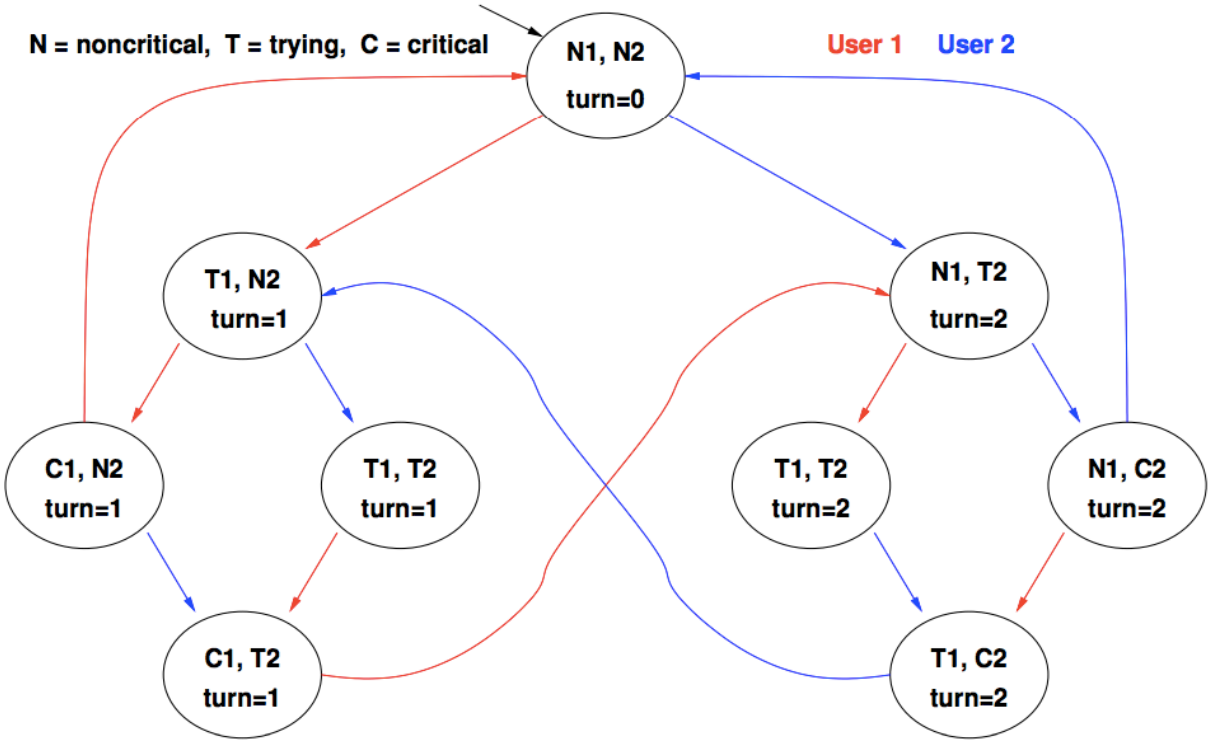
Mutual Exclusion – Safety



$G\neg(C1 \wedge C2)$?

Yes, there is no reachable state in which $C1 \wedge C2$ holds

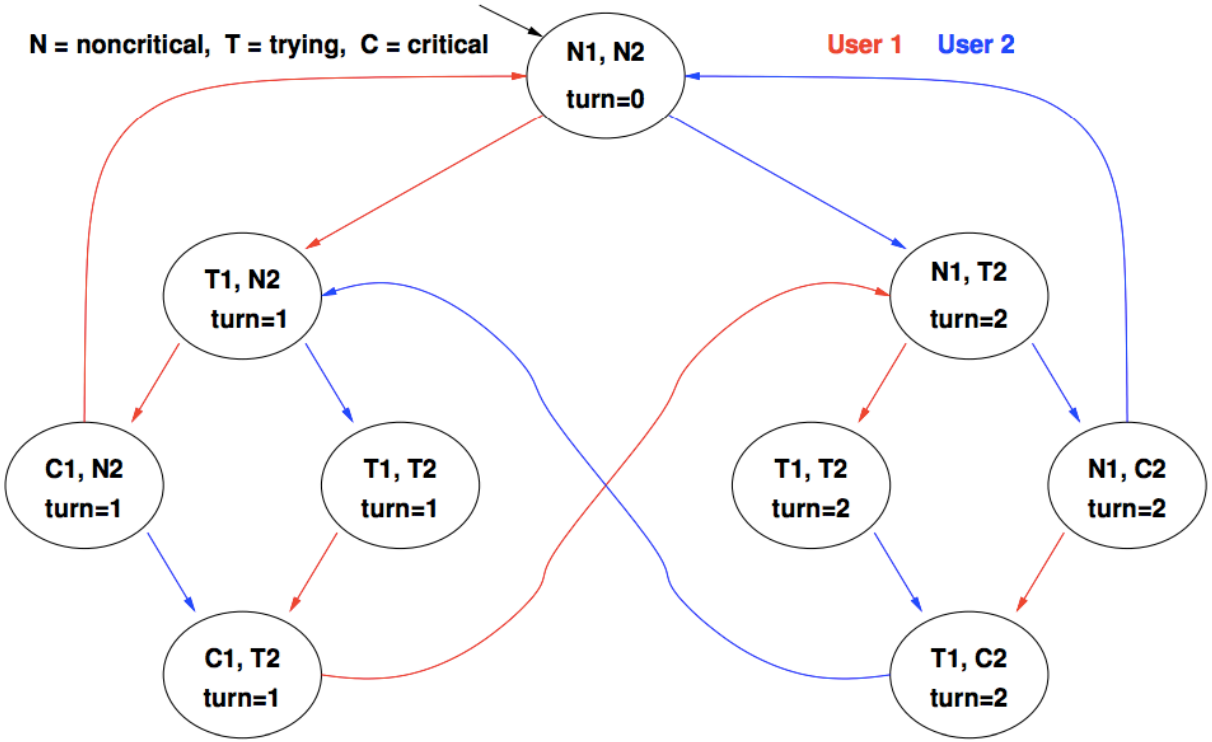
Mutual Exclusion – Liveness (I)



F*C1*?

No, the blue cyclic path is a counterexample

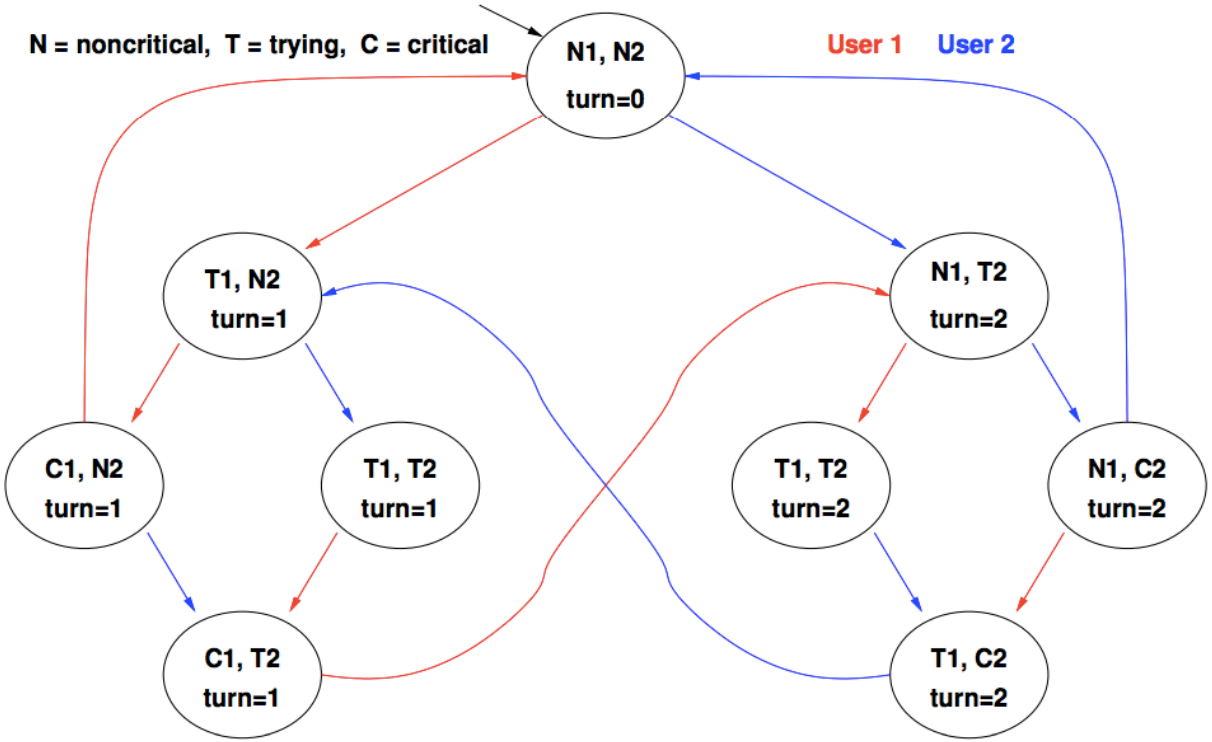
Mutual Exclusion – Liveness (II)



G(*T1*→F*C1*)?

Yes, in every path, if *T1* holds then *C1* holds afterwards

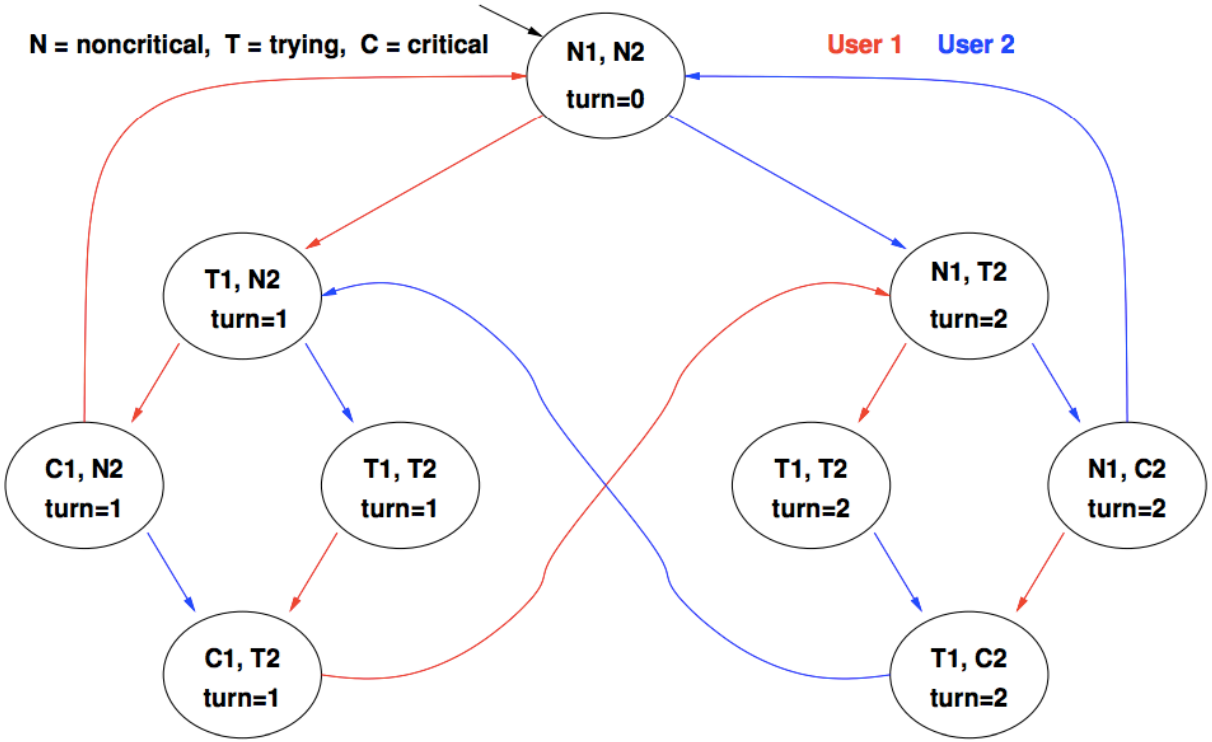
Mutual Exclusion – Liveness (III)



AG(*N1*→EF*T1*)?

Yes, from each state where *N1* holds there is a path leading to a state where *T1* holds

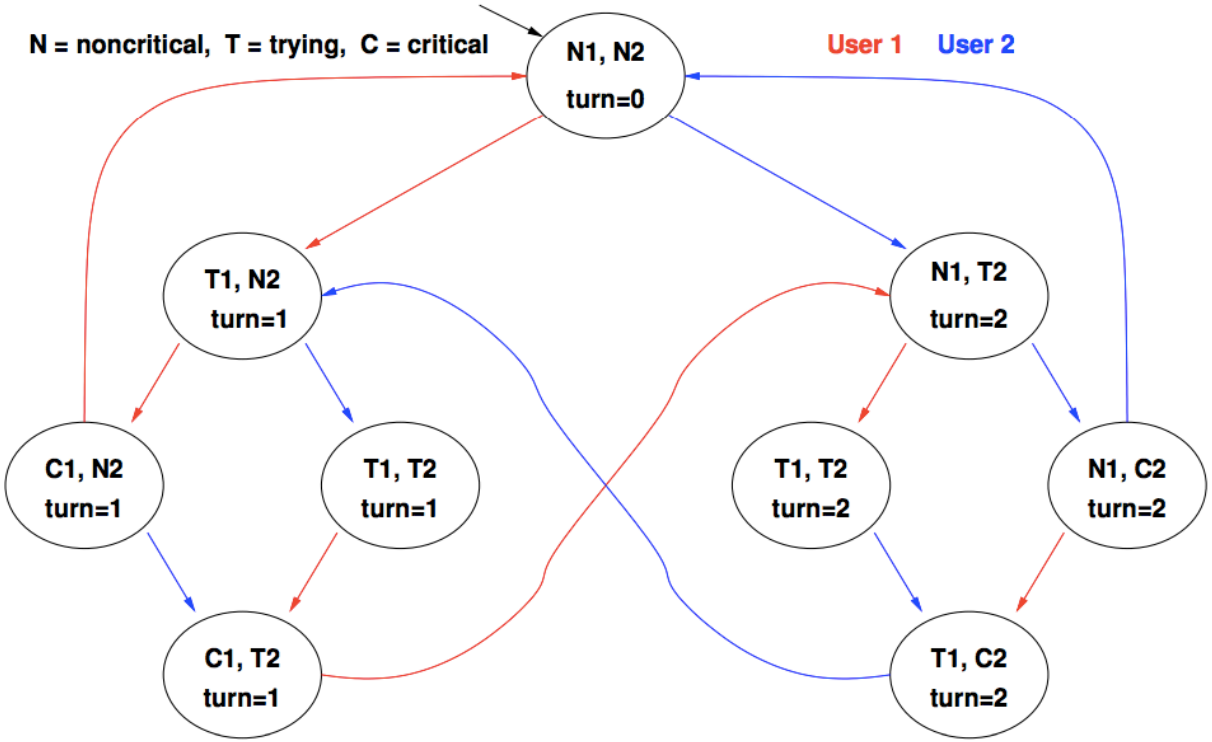
Mutual Exclusion – Fairness



GF*C1*?

No, the blue cyclic path is a counterexample

Mutual Exclusion – Strong Fairness



GF*T1*→GF*C1*?

Yes, every path that visits *T1* also visits *C1*

Concurrent Software Systems in Java

"Sometimes abstraction and encapsulation are at odds with performance—although not nearly as often as many developers believe—but it is always a good practice first to make your code right, and then make it fast."

B. Goetz, *Java Concurrency in Practice*, Addison-Wesley, 2006

Concurrent Software Systems (I)

- A software system is *concurrent* if its computation is obtained by the composition of independently executing sequential computations, which can be virtually executed in parallel under system-specific constraints that impose the sequentialization of parts of the computations
- The computation of a concurrent system is structured in terms of a (possibly dynamic) set of independent *execution flows*
  - Each execution flow performs a sequential computation
  - Execution flows are composed in parallel under sequentialization constraints
  - Execution flows can share the data and the resources needed to perform their computations

Concurrent Software Systems (II)

- The static description of a concurrent software system is provided in terms of a set of *concurrent programs*
  - Each concurrent program describes some of the execution flows of the concurrent system
  - Each concurrent program describes some of the needed sequentialization constraints among execution flows
- The description of the sequential computations associated with execution flows is provided in a program using a *programming language*
- If the adopted programming language allows explicitly describing also the needed sequentialization constraints, then it is a *concurrent programming language*

Concurrent Systems in Java

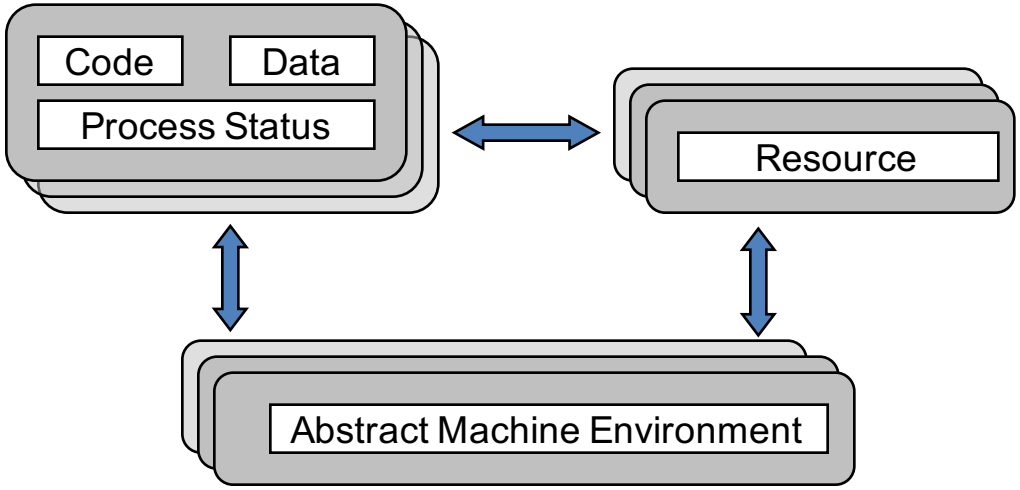
- Java is a concurrent programming language
  - Java provides language constructs to express basic sequentialization constraints
  - Complex patterns of sequentialization can be expressed by means of the combined use of the provided language constructs
  - *The Java platform is designed from the ground up to support concurrent programming, with basic concurrency support in the Java programming language and the Java class libraries. Since version 5.0, the Java platform has also included high-level concurrency APIs.* [The Java Tutorials]

Processes (I)

- A (concurrent) process is a (concurrent) program in execution
  - A program is a static document that describes the execution-time behavior of a process
  - A process is a dynamic entity that executes a program using a particular set of data and resources
  - Two or more processes can execute the same program, each using their own data and resources
- A process comprises, at least, the following components
  - The program to be executed
  - The data on which the program will execute
  - The resources required by the process at execution-time
  - The *status* of the process execution

Processes (II)

- A process executes in an *abstract machine environment* that manages the sharing and isolation of data and resources among a community of processes
- The Java Virtual Machine (JVM) provides the abstract machine environment for Java processes (one JVM for one process)

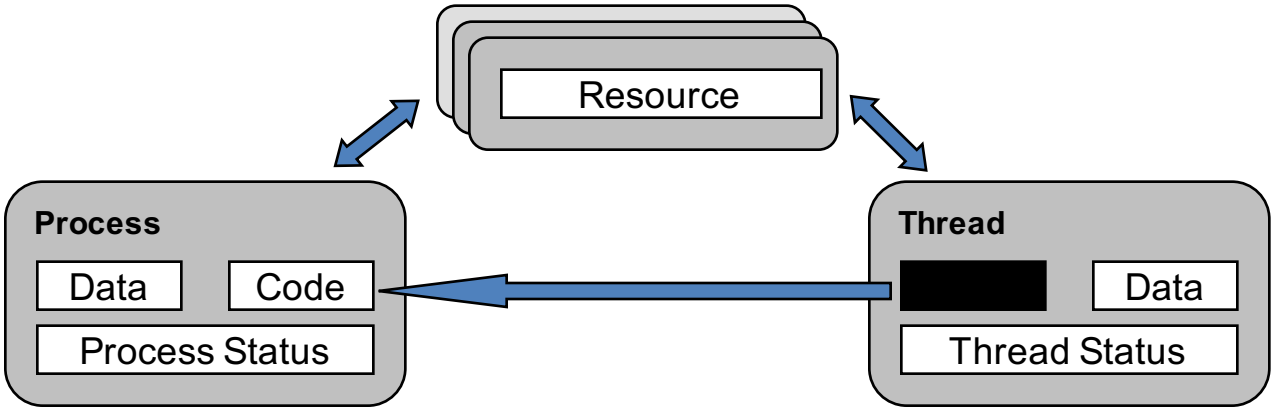


Threads (I)

- Threads (of control, of execution) are the execution flows of a process
  - Each thread is associated with a single process
  - Several threads can be associated with a single process
  - The set of threads associated with a process is dynamic because threads are dynamically started and terminated
  - A thread executes part of the program associated with its process using the data and the resources of the process
  - A thread dynamically allocates part of the resources of the process for its computational needs
  - Besides the data of its process, a thread has its own (private) data and status

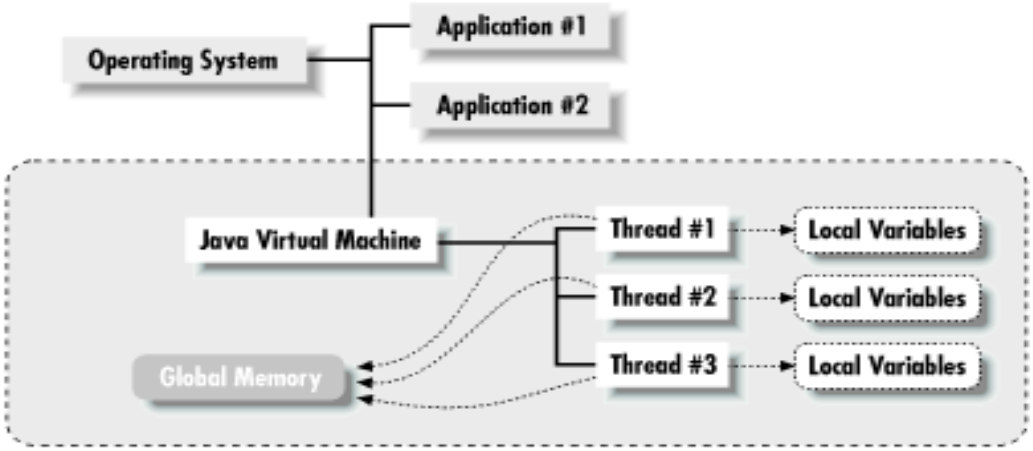
Threads (II)

- Note that
  - A *sequential process* has a single thread
  - A *shared memory* is visible to all the threads of a process
  - Threads are sometimes called *lightweight processes* because the overhead associated with their computation is reduced with respect to that of processes (but a process is always needed to execute threads)



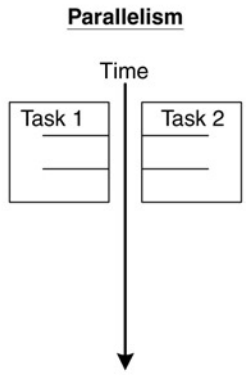
Threads (III)

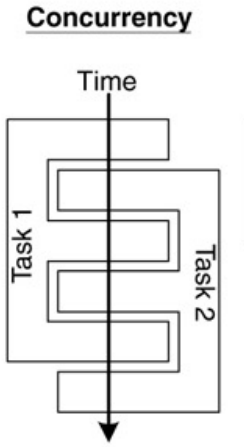
- Threads have the following properties
  - A thread begins its execution at a specific point of the program
    - For one of the threads, called the *main thread*, that point is the *entry point* of the program
    - For each one of the other threads, that point is a specific point of the program, called the entry point of the thread, that is decided at compile time or at runtime
  - A thread executes in an ordered and predefined sequence
  - A thread executes independently of other threads
  - Threads appear to executed in parallel, even if they can be interleaved



Concurrency and Parallelism

- *Parallel systems* are deployed on a set of *CPUs* to execute several processes and their threads in parallel, i.e., at the same time
  - A memory shared among CPUs is normally present
  - If no shared memory is present, and communication uses a network, the system is called *distributed*
- *Concurrent systems* are possibly deployed on a single CPU, but they are structured as if they were parallel systems
  - Sometimes, parts of a concurrent system are actually parallel systems
  - The designer is requested to assume that processes and threads are executed in parallel





Threads in Java (I)

- There are two ways to describe a thread in Java
  - By extending the `java.lang.Thread` class using top-level classes or (anonymous) inner classes
  - By implementing the `java.lang.Runnable` interface using top-level classes, (anonymous) inner classes, lambda expressions, or static method references
- Each thread is associated with an object called *thread object*

```
public class MyThread extends Thread {
    @Override
    public void run() {
        // the body of the thread
    }
}
```

Threads in Java (II)

- There are two ways to describe a thread in Java
  - By extending the `java.lang.Thread` class using top-level classes or (anonymous) inner classes
  - By implementing the `java.lang.Runnable` interface using top-level classes, (anonymous) inner classes, lambda expressions, or static method references
- Each thread is associated with an object called *thread object*

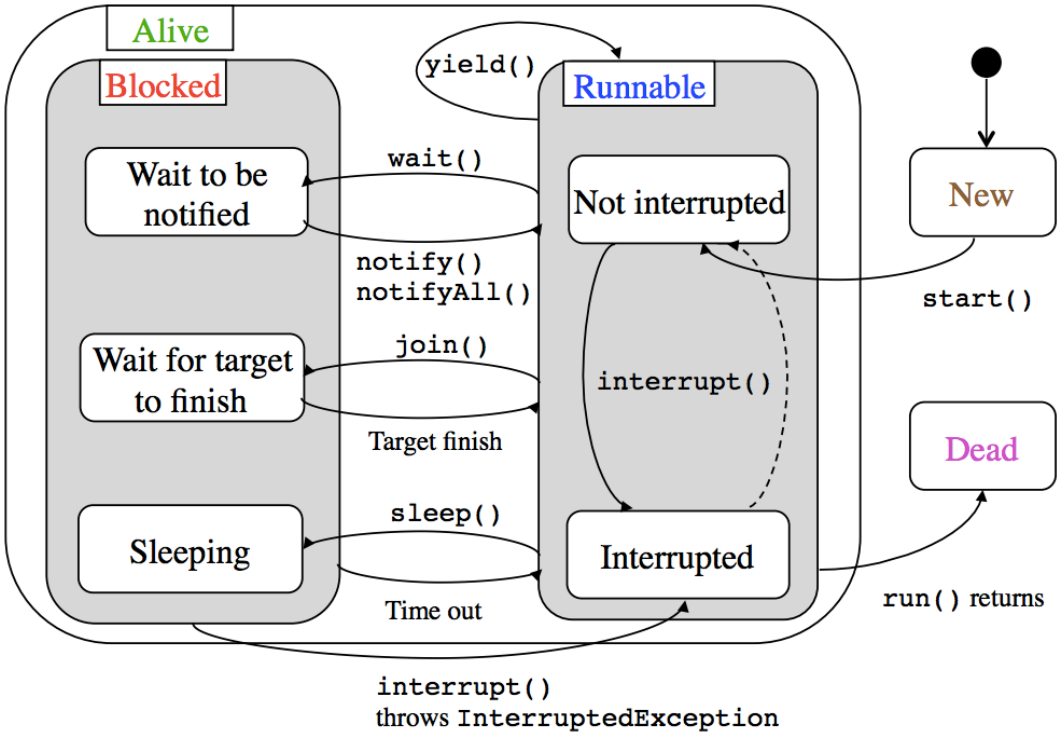
```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        // the body of the thread
    }
}
```

Threads in Java (III)

- In both ways, the `run()` method contains the *section* of the program that the thread executes
  - The beginning of the `run()` method is the entry point of the thread
- The `start()` method is used to actually start a thread and begin the execution of the `run()` method
  - It is called on a thread object
  - For example, by extending `Thread`:
    ```
    new MyThread().start();
    ```
  - For example, by implementing `Runnable`:
    ```
    new Thread(new MyRunnable()).start();
    ```
- A thread terminates when its `run()` method terminates
  - A thread cannot be forcibly terminated

Threads in Java (IV)

- Threads in Java have a *lifecycle state* that is structured in three levels



Threads in Java (V)

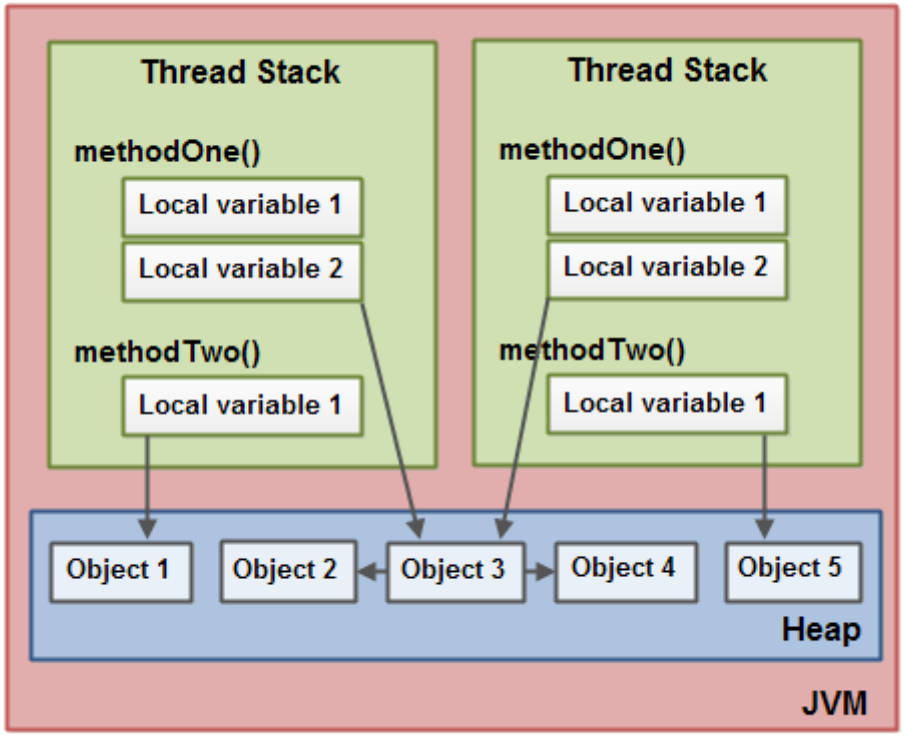
- The lifecycle state of a thread can be altered programmatically
  - The `sleep(millis)` method forces the thread to the blocked state for the specified amount of milliseconds
  - The `interrupt()` method forces the thread to the running state, and if the thread is in the blocked state when `interrupt()` is invoked, the method that forced the thread to the blocked state throws an `InterruptedException`
  - The `interrupted()` method checks whether the thread has been previously interrupted
  - The `yield()` method keeps the thread in the running state, but it calls the thread scheduler to allow other threads to execute
  - The `join()` method forces the thread to the blocked state, if necessary, and it waits for the threads on which it is invoked to terminate

Threads in Java (VI)

- On a single CPU, threads execute one at a time in such a way as to provide the illusion of parallelism
  - However, different threads in the same process can execute on different CPUs to ensure true parallel execution
- The JVM implements a very simple scheduling algorithm for threads called *fixed priority scheduling*
  - The JVM schedules threads based on their priorities relative to other threads in the runnable state
  - The JVM chooses for execution one of the thread in the runnable state with the highest priority
  - If two threads in the runnable state and with the same priority are waiting, the scheduler chooses one of them in *round-robin* order
- Note that the priorities of threads can be decided upon creation, and they can changed whenever necessary

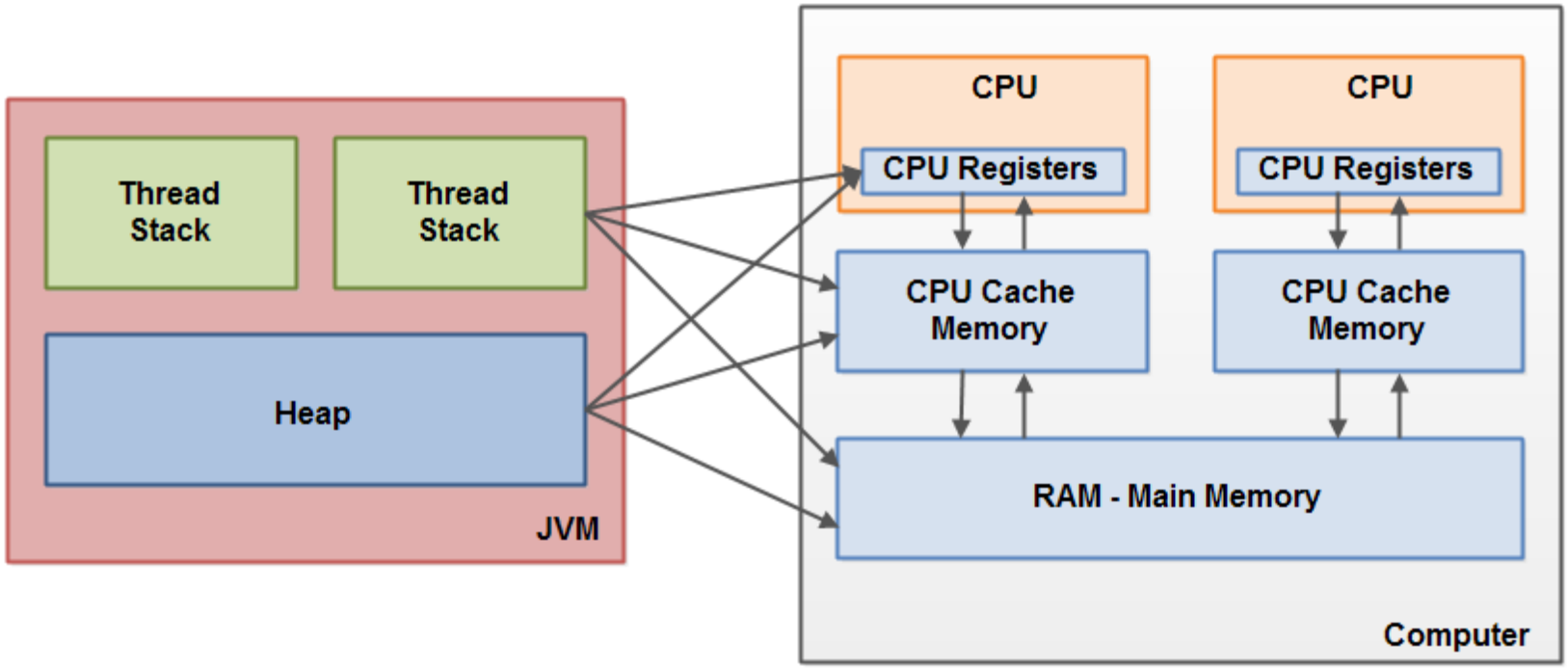
Java Memory Model (I)

- The threads of a Java program
  - Have a private *stack* used to support method invocations
  - Have a private *thread-local storage*
  - Share the *(object) heap*
- The Java memory model
  - Describes how threads access their three types of memories
  - Describes how the content of memories is actually stored in the *memory hierarchy* of the system



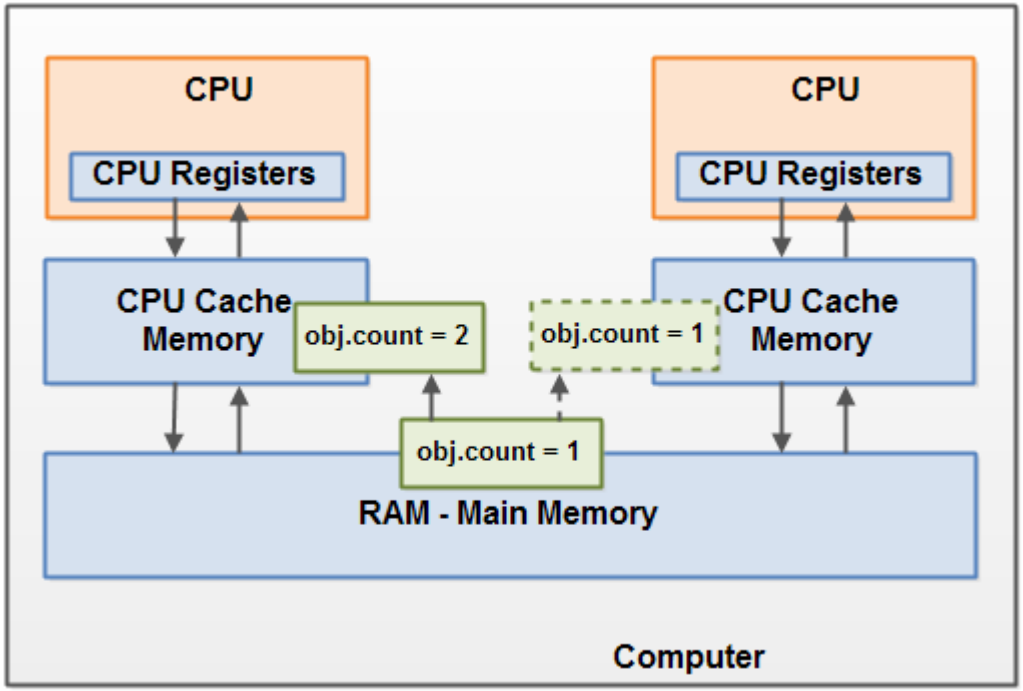
Java Memory Model (II)

- The threads of a Java program are not necessarily executed by the same CPU because the JVM is normally distributed among several CPUs



Java Memory Model (III)

- The execution of threads on multiple CPUs causes relevant *memory-coherence* problems
- Such problems are solved if the access to shared data is controlled
- Note that the needed controlled access can cause the inefficient use of parallelism and of resources



Mutual Exclusion in Java (I)

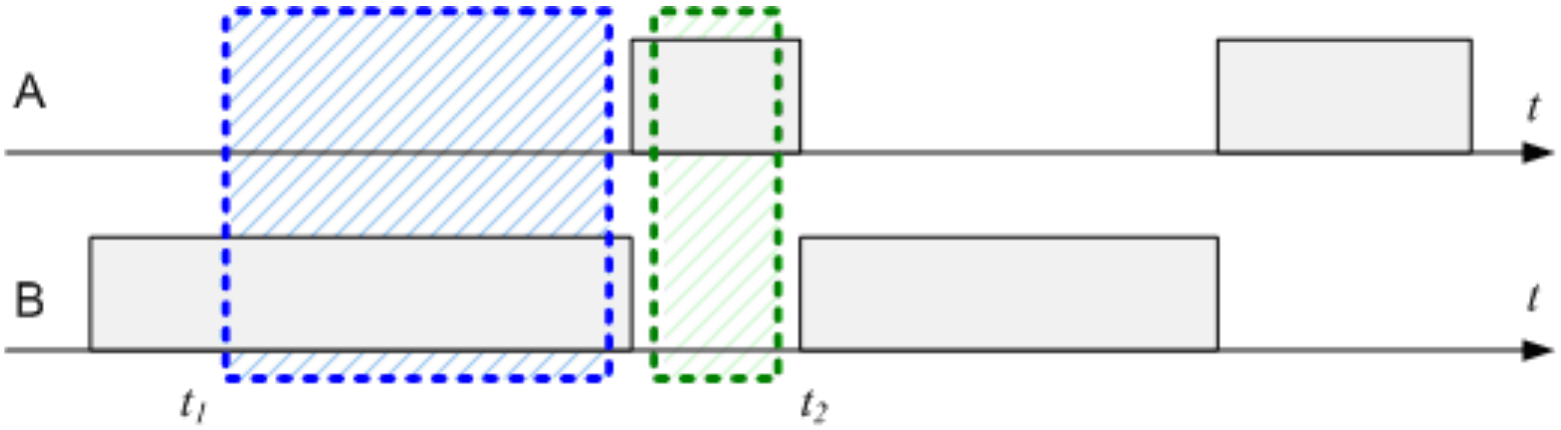
- The problem of mutual exclusion is one of the most basic problems of concurrent programming
- A concurrent program faces a mutual exclusion problem when it must be ensured that if, given a set *M* of *mutually exclusive* sections of a program,
  - Only one thread at a time can execute one of the sections of the program in *M*
  - The threads that cannot execute the sections in *M* are blocked and resumed as soon as possible
- For example, if a program contains a set $M=\{ P_1, P_2, \ldots, P_N \}$ of mutually exclusive procedures, then only one thread at a time can execute one of the procedures in *M*

Mutual Exclusion in Java (II)

- Mutual exclusion is solved in Java using *critical sections*
- A critical section is a section of a program associated with an object used as a MUTual EXclusion device (mutex) to control the access to the critical section
  - A mutex can be *acquired* and *released* by threads, and it is *owned* by a thread after its acquisition and before its release
  - If a thread acquires a mutex, then no other threads can acquire that mutex until it is explicitly released
  - Threads are forced to the blocked state when they cannot acquire a mutex, and they are automatically forced to the running state when they can finally acquire it
  - A thread executes a critical section only when it is owning the mutex used to guard the section

Mutual Exclusion in Java (III)

- Note that acquisition and release are
  - *Reentrant* operations because a mutex can be acquired and released in nested loops several times without blocking the thread that owns the mutex
  - *Synchronous* operations within the process because their effects are synchronously shared among all the threads of the process, possibly among several CPUs



Mutual Exclusion in Java (IV)

- A critical section is identified by
  - The `synchronized` *modifier* of methods, to state that the body of the method is the critical section, and that `this` references the object to be used as mutex
  - The `synchronized` block, to state that the *body* of the block is the critical section, and that the object referenced in the *head* of the block is used as mutex

```
public synchronized void myMethod() {
    // the critial section
}
```

Mutual Exclusion in Java (V)

- A critical section is identified by
  - The `synchronized` *modifier* of methods, to state that the body of the method is the critical section, and that `this` references the object to be used as mutex
  - The `synchronized` block, to state that the *body* of the block is the critical section, and that the object referenced in the *head* of the block is used as mutex

```
public void myMethod(Object o) {
    synchronized(o) {
        // the critial section
    }
}
```

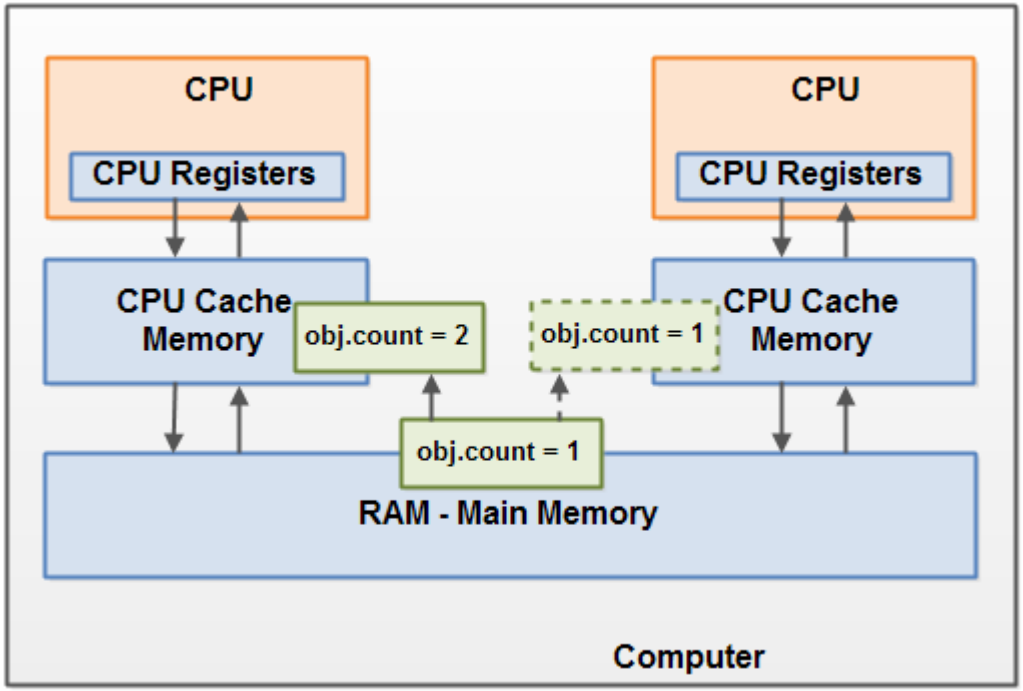
Mutual Exclusion in Java (VI)

- The availability of critical sections in Java provides an effective solution to several problems
  - *Thread interference* is the problem that occurs when two operations, running in different threads, but acting on the same data, *interleave* or are executed in parallel, thus causing a race condition
  - *Memory coherence* is the problem that occurs when different threads have inconsistent views of the copies of the same data across CPUs, e.g., because of unaligned caches

- Note that the use of critical sections to guard against memory coherence problems is a design choice taken for Java
  - Other approaches to solve memory coherence problems that do not require critical sections are possible

Happens-Before (I)

- Java defines the happens-before relation on memory operations such as reads and writes of shared variables
- The results of a write by one thread are guaranteed to be visible to a read by another thread only if the write operation *happens-before* the read operation



Happens-Before (II)

- Each action in a thread *happens-before* every action in that thread that comes later in the program
- An unlock of a mutex *happens-before* every subsequent lock of the same mutex
  - The *happens-before* relation is transitive, and therefore all actions of a thread prior to unlocking *happen-before* all actions subsequent to any thread locking that mutex
- A write to a `volatile` field *happens-before* every subsequent read of that same field
  - Note that the volatile modifier does not entail mutual exclusion
- A call to `start()` on a thread *happens-before* any action in the started thread
- All actions in a thread *happen-before* any other thread successful return from a `join()` on that thread

Happens-Before (III)

- The `synchronized` keyword (in either form) can be used effectively to solve memory coherence problems

- The use of the `synchronized` keyword (in either form) on an object `obj` ensures that all changes to the state of `obj` are propagated to all interested threads before any subsequent synchronized access to `obj`
  - Therefore, the use of the `synchronized` keyword solves memory coherence problems
  - Note that `final` fields, which cannot be modified after the object is constructed, can be safely read once the object is constructed (the fields can be safely read, not the referenced objects!)

Happens-Before (IV)

- *Atomic operations* in Java are read and write operations that cannot be interrupted to suspend the current thread and activate another thread
- In detail
  - Reads and writes are atomic for references and for most primitive variables (all types except `long` and `double`)
  - Reads and writes are atomic for all variables and fields declared `volatile` (including `long` and `double` variables and fields)
- Note that
  - Atomic operations do not suffer from thread interference problems
  - However, memory consistency problems are still possible if the `volatile` modifier is not used

Waiting and Notifying Events in Java (I)

- The problems of waiting and notifying events is another one of the most basic problems of concurrent programming
- A thread faces the problem of waiting for events if it needs to wait for events without actively checking for the occurrence of events (*busy waiting*)
- A thread faces the problem of notifying events if it needs to communicate the occurrences of events to another thread, which is possibly waiting for notifications
- Java couples the problem of waiting and notification of events to critical sections because critical sections already provides a means to block threads (without busy waiting)

Waiting and Notifying Events in Java (II)

- A thread that needs to notify events
  - First, it enters a critical section guarded by an object `obj`
  - Then, it invokes the `notifyAll()` method on `obj` to notify all objects that are waiting on `obj` that an event occurred
  - Finally, it proceeds and eventually leaves the critical section

- A thread that needs to wait for events
  - First, it enters a critical section guarded by an object `obj`, which is the same object used to notify events
  - Then, it invokes the `wait()` method on `obj` to block and start waiting for events
  - Finally, after returning to the running state because of a notification on `obj`, it proceeds and eventually leaves the critical section

Waiting and Notifying Events in Java (III)

- Note that
  - The `notifyAll()` method notifies all waiting threads, which are all forced to the running state even if only one of them would eventually proceed in the critical section
  - The method `notify()` method notifies just one of the waiting threads, which is the thread that would eventually proceed in the critical section
  - The use of the `notify()` method is inherently more efficient, but it is also error prone

Waiting and Notifying Events in Java (IV)

- Note that
  - The `wait(millis)` method can be used to wait for no more than `millis` milliseconds (*best effort deadline*)
  - The `wait(millis,nanos)` method can be used to wait for no more than `millis` milliseconds and `nanos` nanoseconds (*best effort deadline*)
  - The `wait()` methods and its variations throw an `InterruptedException` if any thread interrupted the current thread before or while the current thread was waiting for an event
    - When an `InterruptedException` is thrown, the thread is forced to the state non-interrupted
    - The same behavior is adopted by the `sleep()` and the `join()` methods

Liveness Problems in Java

- The superficial or erroneous use of the support for concurrent programming that Java provides can result in severe liveness problems
- Some of such problems are
  - *Deadlock*, which is a situation in which two or more threads are blocked forever, waiting for each other
  - *Livelock*, which is a situation in which two threads continuously react to the events that each one of them notifies to the other
  - *Starvation,* which is a situation in which a thread is unable to gain regular access to a shared resource and, therefore, is unable to make the expected progress

High-Level Abstractions for Concurrency

- Java provides basic mechanisms to manage concurrency and its inherent problems
  - High-level *abstractions* are built using such mechanisms
- High-level abstractions to manage the problems related to concurrency are needed
  - To improve the maintainability of concurrent systems
  - To improve the reusability of solutions to concurrency problems
  - To improve the level of understanding of the nonfunctional characteristics of solutions (e.g., level of parallelism, type of control, liveness properties)

Blocking Queues (I)

- A *queue* is a sequence of items that changes dynamically according to the *FIFO (First In, First Out)* policy
- Basic operations on queues are
  - *Creation*, to create an empty queue
  - *Destruction*, to destroy a queue
  - *Is empty test*, to check if a queue is empty
  - *Is full test*, to check if a queue is full
  - *Enqueue*, to add an item to the queue
  - *Dequeue*, to remove an item from the queue

Blocking Queues (II)

- A *blocking queue* is a queue intended for concurrent use
- The operations block if they cannot be performed immediately
  - *Enqueue* can block if the queue is full
  - *Dequeue* can block if the queue is empty
- Note that
  - All blocking queues can block on dequeue if they are empty
  - Only blocking queues with limited capacity can block on enqueue if they are full
- Blocking queues are an abstraction that can be used to *coordinate* the activities within a concurrent system

Locks and Conditions (I)

- An *(explicit) lock* is an abstraction that can be used to ensure mutual exclusion
  - Overcomes inherent limitations of critical sections regarding the nesting of sections
  - Is (much more) error prone than critical sections
- A lock can be
  - Explicitly *locked* (*acquired*) and *unlocked* (*released*)
  - Only one thread at a time can *own* the lock
  - A thread can block in the attempt to acquire a lock that is already locked by another thread
  - A thread that blocked in the attempt to acquire a lock is restarted as soon as the lock can be acquired

Locks and Conditions (II)

- A *condition* is an abstraction that can be used to *wait* for interesting events and to *signal* interesting events
  - A thread can signal that a condition has become true
  - A thread can block waiting for a condition to be signaled
  - A lock is always needed to guard a *condition*
- A lock is normally associated with several conditions
  - Waiting and signaling on conditions is possible only for the thread that owns the lock of the conditions
  - When a thread waits for a condition to be signaled, it releases the lock of the condition
  - When a thread signals a condition, it needs to explicitly release the lock of the condition

Atomic References

- An atomic reference encapsulates a reference to an object and manages it in mutual exclusion
- The following operations are normally provided
  - Deference the atomic reference (read the embedded reference)
  - Assign the atomic reference (write the embedded reference)
  - Deference the atomic reference and assign it (read the embedded reference and substitute it)
  - Deference the atomic reference to *R* and assign it to *f(R)*, returning *R* (read the embedded reference and update it)
  - Deference the atomic reference to *R* and assign it to *f(R)*, returning *f(R)* (read the embedded reference and update it)
- Previous operations are all performed *atomically*

Pools of Resources

- Concurrency problems are often caused by *shared resources*
  - The correct management of the access to shared resources is one of the most basic concurrency problems
- A group of identical resources is called a *pool of resources*
  - When the pool is created/destroyed, all resources are also acquired/released (but resources can be released before)
  - Resources are assigned for usage upon request to the pool
  - The controlled access to the resources is guaranteed by the pool
- Pools of resources are normally used to control the amount of resources needed by a concurrent system
  - To ensure that a sufficient amount of resources is available
  - To ensure that resources are efficiently used

Thread Pools

- Threads are virtually the most relevant resources in a concurrent system
- The construction, destruction, and access to threads are often controlled using *(simple) thread pools*
- When it is created, a thread pool creates and activates all threads in the pool
  - To ensure that threads are immediately available when needed
  - To ensure that the level of concurrency is controlled
- More complex thread pools are sometimes used

Executors (I)

- A *(simple) executor* is an abstraction that can be used to concurrently perform tasks
  - It is associated with a thread pool used to perform the tasks
  - It enqueues the tasks that cannot be started immediately
  - It provides ways to return the results of tasks (if any)
  - It provides ways to return the exceptions that caused the termination of tasks (if any)
  - It allows interrupting tasks and gracefully terminating all threads in the pool
  - (Sometimes) It provides a set of scheduling policies

Executors (II)

- An executor provides three ways to execute tasks
  - *One way execution*, for which the executor does not provide a means to known if the task was actually executed and to read the result of the task (if any)
  - *Execution with callback*, for which the executor allows a *callback task* to use the result of the task (if any), often in the thread that executed the requested task
  - *Execution with future*, for which the executor provides a *future* (a *promise*) that handles the result of the requested task (if any) when it becomes available
- Executors provide for *asynchronous tasks*

Callbacks

- A *callback task* is a task that is executed when an executor completes a requested task
  - The callback task receives the result of the requested task (if any)
  - The callback task receives the exception that caused the termination of the requested task (if any)
  - (Sometimes) The callback task is executed in the same thread that executed the requested task
- Callback tasks are implemented as objects that are associated with a request for execution of a task

Futures (I)

- A *future* (or *promise*) is a means to handle
  - The result of an asynchronous task (if any)
  - The exception that caused the termination of the asynchronous task (if any)
- Futures are implemented as objects that are associated with a request for execution of a task
  - They are dynamically associated with the result of the requested task (if any) when it becomes available
  - They are dynamically associated with the exception that caused the termination of the requested task (if any) when it becomes available

Futures (II)

- A future blocks the thread that tries to access its embedded value (a result or an exception) if the value is not yet available
  - The thread is resumed when the result becomes available (if any), and the result is returned
  - The thread is resumed when an exception becomes available (if any) , and the exception is thrown
- Note that a future do not block if its value is already available when it is requested

Java Reflection (I)

- Java and the *Java Virtual Machine* (*JVM*) provide a means, the `java.lang.reflect` package, to postpone some decisions at runtime
- Java remains a *statically typed* language, but it provides a pure object-oriented means to support
  - Dynamic *linking* of classes
  - (Dynamic) *Introspection* (of objects)
  - Dynamic *creation* of objects
  - Dynamic *access* to fields
  - Dynamic *invocation* of methods

Java Reflection (II)

- Note that the following facts are true in Java
  - Every object is associated with the class that was used to create it, the so called *factory class*
  - Every class/interface is represented at runtime by an object, the so called *class object* (or *class descriptor*)
    - Given class/interface `C`, the JVM provides one object of class `java.lang.reflect.Class<C>`
    - Given class/interface `C`, the object that represents the class/interface can be referenced by `C.class`
  - Every class object is associated with a *class loader*, which is the object that was used to load the bytecode of the class
- Class objects are the entry point of Java reflection

Class Objects (I)

- The following are some ways to obtain a class object
  - Given an object `o`, `o.getClass()` returns the class object associated with `o`
  - Given a string `n` containing the full name of a class, `Class.forName(n)` returns the right class object
  - Given a class loader `l` and a string `n` containing the full name of a class, `l.loadClass(n)` returns the right class object
- Note that class objects can be accessed by name even if no objects of the class are available
  - `ClassNotFoundException` may be thrown

Class Objects (II)

- Beside the functionality related to dynamic Java reflection, an object `c` of class `Class<C>` can
  - Perform a type cast of object `o` to the represented class, with `c.cast(o)`, which is equivalent to `(C)o`
  - Check if object `o` is an instance of the represented class, with `c.isInstance(o)`, which is equivalent to `o instanceof C`
  - Check if the referenced class/interface is the same or is a superclass/superinterface of a class represented by `k`, with `c.isAssignableFrom(k)`
- A few other uses of class object are not discussed

Introspection

- Given an object `c` of class `Class<C>`, it is possible to *inspect* the structure of `C` and, for example,
  - It is possible to list the *field descriptors* of the visible fields of `C`
  - It is possible to list the *constructor descriptors* of the visible constructors of `C`
  - It is possible to list the *method descriptors* of the visible methods of `C`
  - It is possible to obtain a reference to the class object of the base class (or superclass) of `C`
  - It is possible to obtain references to the class objects of the interfaces that `C` implements
- The word *introspection* refers to the possibility of an object to inspect its class

Dynamic Object Creation

- Given an object `c` of class `Class<C>`, it is possible to create objects of class `C`, for example,
  - Using `c.newInstance()`
  - Using `c` to access one of the constructor descriptors of `c` and invoking the constructor with suitable arguments
- Note that, if `C` is known (it is not `?` or a type parameter), then the dynamic object creation does not require explicit type casts
  - For example, the following is an empty string

```
String s = String.class.newInstance()
```

Dynamic Access to Fields

- Given an object `c` of class `Class<C>`, it is possible to access the field descriptors of the visible fields of `c`
- Besides describing fields, field descriptors can be used to get or set the value of the field for some object
- Given an object `c` of class `Class<C>` , a field descriptor `f` obtained by `c`, and an object `o` of class `C`, it is possible to
  - Use `f.get(o)` to read the current value of the field for `o`
  - Use `f.set(o, v)` to set the current value of the field to `v` for `o`
- Note that class `java.lang.reflect.Field` is not generic, and therefore, it does not provide the compiler with the type of the described field

Dynamic Method Invocations

- Given an object `c` of class `Class<C>`, it is possible to access the method descriptors of the visible methods of `c`
- Besides describing methods, method descriptors can be used to invoke the described method with suitable arguments
- Given an object `c` of class `Class<C>`, a method descriptor `m` obtained by `c`, an object `o` of class `C`, and an array of objects `a`, it is possible to use `m.invoke(o, a)` to invoke the method described by `m` on object `o` with arguments `a`
- Note that class `java.lang.reflect.Method` is not generic, and therefore, it does not provide the compiler with (for example) the return type of the described method

Dynamic Proxies

- Given an array `a` of class objects associated with interfaces, a *dynamic proxy* is an object that implements the interfaces in `a` and invokes user code upon method invocations
- Note that
  - Proxies are created using `java.lang.reflect.proxy.Proxy.newInstance`
  - The user code invoked upon method invocation is arbitrary, and it is an implementation of the functional interface `java.lang.reflect.InvokationHandler`
  - The user code is provided with the descriptor of the invoked method
  - The user code can return a value, which is used as the return value of the invocation that triggered the activation of the user code

Aspect-Oriented Programming (I)

- *Aspect-Oriented Programming* (*AOP*) has been advocated as the next step after *Object-Oriented Programming* (*OOP*) since the early 2000s (or even earlier)
  - It is primarily intended to promote reuse
- In a simplistic picture, AOP regards adding *aspects* to OOP
  - Several approaches exist, and some approaches even extend beyond OOP to reach other *programming paradigms*
- In the scope of the (quasi-)pure OOP that Java advocates, AOP can be drastically simplified
  - Aspects are features of objects that are not readily provided by their classes
  - An *aspect provider* is an object that can *attach*/*detach* an aspect to an object, or that can create an object with a requested aspect

Aspect-Oriented Programming (II)

- In this peculiar view of AOP, the interest is in providing implementations of aspects that are
  - (Mostly) Independent from the characteristics of the objects to which they are attached
  - (Almost) Freely composable, so that an object can be attached to several aspects, thus acquiring several features
  - (Almost) Orthogonal, so that the composition of aspects provides the sum of independent features
- This particular incarnation of AOP can be obtained in Java using dynamic proxies
  - Given an object ○, an aspect provider attaches the aspect to ○ by means of a dynamic proxy that intercepts all invocations to the public methods of ○

Aspect-Oriented Programming (III)

- For example, the following are some general-purpose aspects that are normally considered
  - *Shared*: a shared object is an object that must ensure mutual exclusion for the execution of its methods
  - *Logging*: a logging object is an object that traces the invocations to its methods in a message log
  - *Persistent*: a persistent object is an object that survives to the shutdown of the system in which it was created or modified
  - *Active*: an active object is an object that executes its methods in a dedicated thread pool
  - *Remote*: a remote object accepts method invocations from remote clients and provides return values and exceptions to its clients
  - *Reloadable (class)*: a reloadable class can be hot swapped at runtime
- Normally, only methods from implemented interfaces are considered when implementing aspects

Shared Aspect

- A shared object is an object that must ensure mutual exclusion for the execution of its methods
- Only methods from implemented interfaces are interesting because those are the exported methods
- A dynamic proxy is sufficient to intercept all invocations to interesting methods
  - The invocation handler provides an objects used as synchronization lock
  - The invocation handler enters a critical region guarded by the lock before invoking to the target method, and it exits the critical region immediately after
  - The invocation handler exits the critical region also in case of exceptions

Logging Aspect

- A logging object is an object that traces the invocations to its methods in a message log
- Only methods from implemented interfaces are interesting because those are the exported methods
- A dynamic proxy is sufficient to intercept all invocations to interesting methods
  - The invocation handler logs before and after invoking the target method
  - The invocation handler logs also in case of exceptions

Persistent Aspect

- A persistent object is an object that survives to the shutdown of the system in which it was created or modified
- Dynamic proxies are not needed because the user is requested to explicitly
  - Commit changes to the persistent store
  - Rollback changes (by reloading data from the persistent store)
- Simple persistence can be obtained by loading/saving *serializable objects* to files
  - Objects that implement `java.io.Serializable` can be easily serialized with `java.io.ObjectOutputStream` and deserialized with `java.io.ObjectInputStream`
  - Serializable objects provide a private `serialVersionUID` field to disambiguate different versions of their classes

Active Aspect (I)

- An active object is an object that executes its methods in a dedicated thread pool
- Only methods from implemented interfaces are interesting because those are the exported methods
- A dynamic proxy is sufficient to intercept all invocations to interesting methods
  - But, if the target object implements interface `T`, another interface `A`, called *active interface*, is needed
  - The active interface provides methods with signatures similar to the signatures of the methods in `T`, but results are returned using futures and callbacks

Active Aspect (II)

- Normally, the active interface `A` of interface `T` is requested to extend `Active<T>`, and if

$$\texttt{R m}(\texttt{T}_1,\texttt{T}_2,\ldots,\texttt{T}_N)$$

is a method of `T`, then

$$\texttt{Future<R> m}(\texttt{T}_1,\texttt{T}_2,\ldots,\texttt{T}_N)$$

$$\texttt{void m}(\texttt{T}_1,\texttt{T}_2,\ldots,\texttt{T}_N,\texttt{Callback<R>})$$

are methods of `A`

Remote Aspect (I)

- A remote object accepts invocations from remote clients and provides return values and exceptions to its clients
- Normally,
  - Remote objects wait for remote requests from clients on a communication channel
  - The same communication channel used to receive requests is also used to send responses back
  - Objects send over the communication channel are serializable
  - Requests contain the identifier of the method and its arguments
  - Responses contain a return value or an exception
- The simplest communication channel for remote objects are *(TCP/IP) sockets* associated to single interactions

Remote Aspect (II)

- A *server (object)* creates a socket to accept incoming requests for connections using `java.net.ServerSocket`
  - The needed *(TCP) port* is passed to the constructor
  - The method `accept()` is used to wait and get an object of class `java.net.Socket` to read from and write to the connected peer
- Normally, a server processes connections in separate threads (taken from a pool), one for each active connection
- A *client (object)* creates a socket to ask for a connection to a server using `java.net.Socket`
  - The *host name* and the port of the target server are passed to the constructor
  - The socket can be used to read from and write to the connected peer

Remote Aspect (III)

- The remote aspect provider can be used to
  - Register an object as a server on a specified port
  - Provide a proxy to send requests to a server and receive corresponding responses
- If the server implements interface `T`, then the dynamic proxy used to remotely access the server implements interface `T`
- Note that a the proxy provides *synchronous method calls* to the server
  - The client waits until the response of a request is available
- *Asynchronous method calls* can be easily obtained by attaching the active aspect to a remote proxy

Reloadable (Class) Aspect

- It is known that class loaders are used to
  - Load the bytecode of classes in memory
  - Create class descriptors and make classes available to the JVM
- The following facts are true in Java
  - Class loaders cannot unload classes
  - Classes loaded from different class loaders are different, even if their fully qualified names equal
- To allow classes to be hot swapped
  - A new class loader is needed for any swap
  - Class dependencies must be also swapped
- The instances of a reloadable class are *reloadable objects*, even if single objects are not reloaded
  - The factory class for an object cannot be changed

Test-Driven Development

- The costs related to software are often the predominant part of the costs of a system
  - The costs related to software are often much more relevant than the costs related to hardware
- The most significant part of the costs related to software are associated with *maintenance* rather than with development
  - Also because maintenance costs are spread in the several years in which the system is used
- One of the major goals of software engineering is to reduce maintenance costs by means of structured reuse of (tested and documented) software frameworks and libraries
  - Test driven development is introduced to emphasize that no parts of a software system can be reused if they were not adequately tested

Costs of Software Systems

- Normally, the costs of a software system are divided into
  - *Direct costs*, which are directly associated with the activities performed for the system
    - The costs of developers
    - The costs of development tools and external libraries and frameworks
    - …
  - *Indirect costs*, which are associated with all the activities needed to support the activities performed for the system
    - The costs of administrative consultants
    - The costs of legal consultants
    - …
- Normally, indirect costs are between 50% (for SMEs) and 100% (for large enterprises) of direct costs

Evolution and Maintenance (I)

- Software systems must evolve because
  - The initial requirements were not captured correctly
  - The requirements changed during the lifecycle of the system
  - …
- The evolution of a software system is inevitable even if the initial requirements were captured correctly and the initial development produced a good product
- Maintenance activities are performed to evolve a software software system concurrently with its use to
  - Remove anomalies (*corrective maintenance*)
  - Improve relevant qualities of the system (*perfective maintenance*)
  - Adapt to the changes of the environment (*adaptive maintenance*)

Evolution and Maintenance (II)

- Maintenance costs are often more than 50% of the costs related to a software system in its entire lifecycle
  - They are often close to 75%
  - They are so relevant also because of the long period of time in which the system is operational
- Maintenance costs are normally spread as follows
  - Corrective maintenance costs: normally about 20%
  - Perfective maintenance costs: normally about 60%
  - Adaptive maintenance costs: normally about 20%
- Therefore, much effort must be spent to reduce perfective maintenance costs by ensuring accurate analysis and testing of a software systems before its deployment

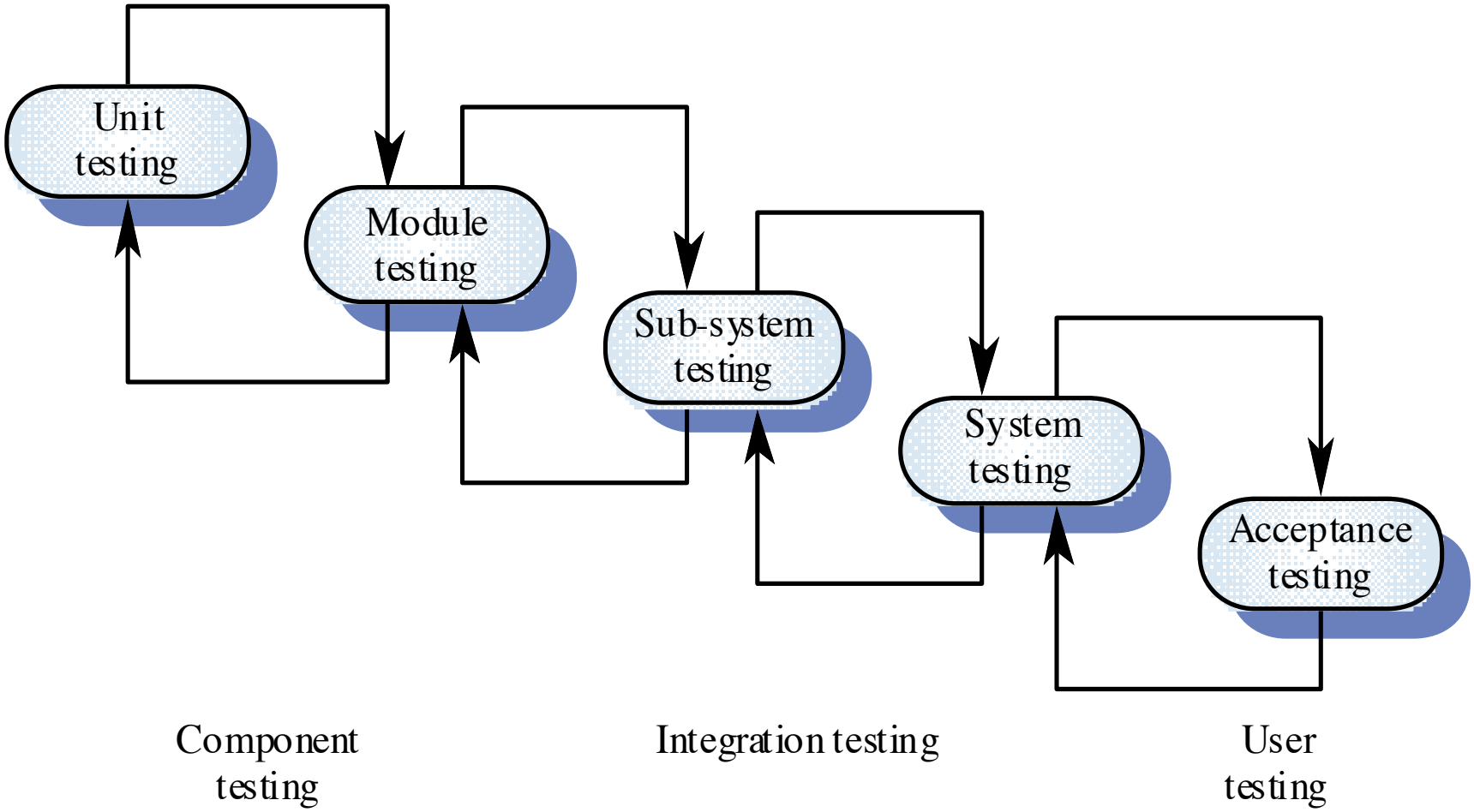
Evolution and Maintenance (III)

- Well-known studies on maintenance costs in deployed systems provide empiric evidence that most of the anomalies could be found by systematic revisions of project *artifacts*
  - In particular, structured and accurate testing is a good way to ensure that anomalies do not propagate to deployed systems
- The following data are interesting to quantify the need of structured and accurate testing
  - 1 out of 10 anomalies found during testing propagates to the deployed system
  - The cost of fixing an anomaly increases by a factor of 10 if the system is deployed
  - Any delay in finding an anomaly impacts severely on the cost of fixing the anomaly

Testing (I)

- Note the following true facts about testing
  - If the behavior of a part of a system is tested in a sufficiently large number of cases, then the behavior of that part can be considered acceptable also in the (hopefully, few) remaining cases
  - Testing can find anomalies, but it cannot prove that a part of a system is correct
- Normally, testing is divided into
  - *Testing in the small*, when single parts are tested
  - *Testing in the large*, when the system as a whole is tested

Testing (II)


Unit testing
Module testing
Sub-system testing
System testing
Acceptance testing
Component testing
Integration testing
User testing

Testing (III)

- Unit testing
  - Testing of single units of reuse
- Module testing
  - Testing of *modules*, which are groups of interdependent units
- Sub-system testing
  - Testing of *sub-systems*, which are small systems that are needed by the system
- System testing
  - Testing of the integration of sub-systems
- Acceptance testing
  - Testing performed together with clients to ensure that the delivered system exhibits the requested technical qualities

Testing in the Small & in the Large

- Testing in the large treats the system as a *black box*, and the testing is intended to check that the behavior of the system is as expected
  - The set of test cases is selected using the artifacts produced to specify the *requirements* of the system
- Testing in the small treats the system as a *white box*, and it examines a sufficiently small part of the system by inspecting the delivered code
  - The set of test cases is selected using the considered code
  - Normally, it is divided into
    - *Statement testing*
    - *Branch testing*
    - *Branch and condition testing*

Statement Testing

- Statement testing is often called *coverage testing* because it is based on the fact that no parts of the code can be considered as tested if they were not executed
  - An anomaly can be found only if the parts of the code that produce it are executed at least once
- A set of test cases *T* can be used to perform the statement testing of a code *C* if, after performing all the test cases in *T*, all statements in *C* were executed at least once

Branch Testing

- Branch testing is often called *path coverage testing* because it is targets the analysis of all the execution paths of the considered code
  - Anomalies are often caused by intricate execution paths that cannot be easily reduced to simpler execution paths
- A set of test cases *T* can be used to perform the branch testing of a code *C* if, after performing all the test cases in *T*, all the execution paths in *C* were executed at least once

Branch and Condition Testing

- Branch and condition testing is often called *condition coverage testing* because it is targets the analysis of the causes that generate different execution paths in the considered code
  - Anomalies are often caused by intricate execution paths that are caused by several interrelated conditions
- A set of test cases *T* can be used to perform the branch and condition testing of a code *C* if, after performing all the test cases in *T*, all the conditions in *C* were considered taking into account all the causes for their truth values

JUnit

- JUnit is an ordinary tool to support structured and accurate testing of Java code
- JUnit works on
  - *Test (cases)*
  - *Test suites*
- Eclipse provides direct support for JUnit to promote test-driven development

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class Tests {
  @Test
  public void getNameNotNull()  {
    Person tester = new Person();

    assertNotNull(tester.getName());
  }

  @Test(expected=IllegalArgumentException.class)
  public void setNameNotNull()  {
    Person tester = new Person();

    tester.setName(null);
  }
}
```

UML

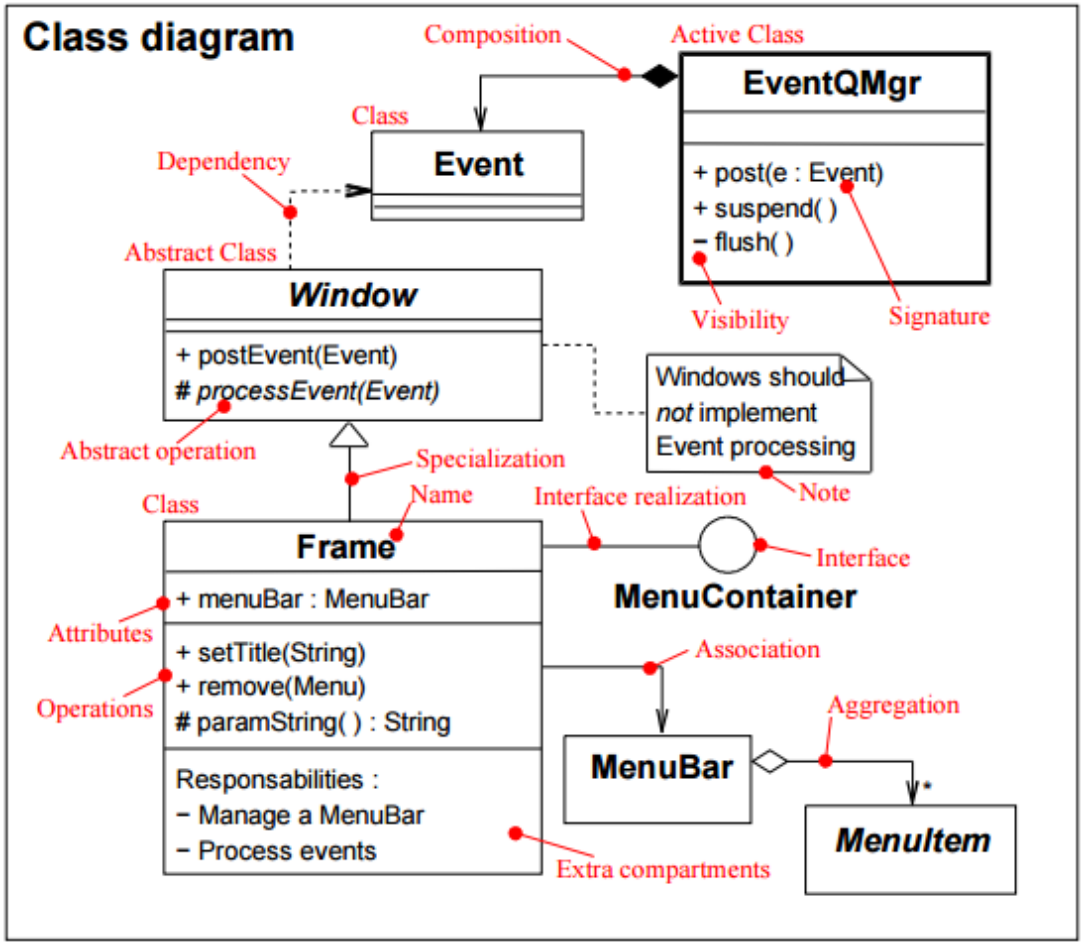
- *UML* (*Unified Modeling Language*)

  `www.uml.org`

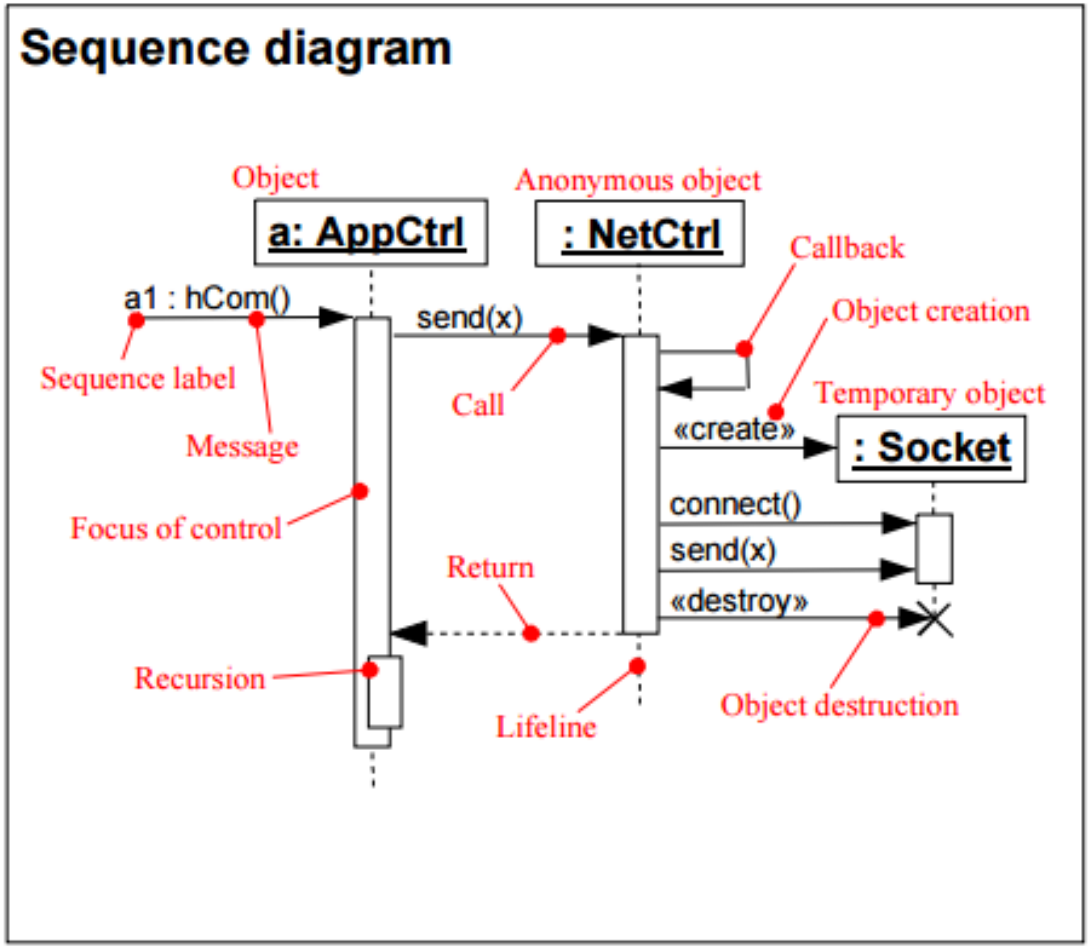
- A standard specified by *OMG* (*Object Management Group*)

  `www.omg.org`

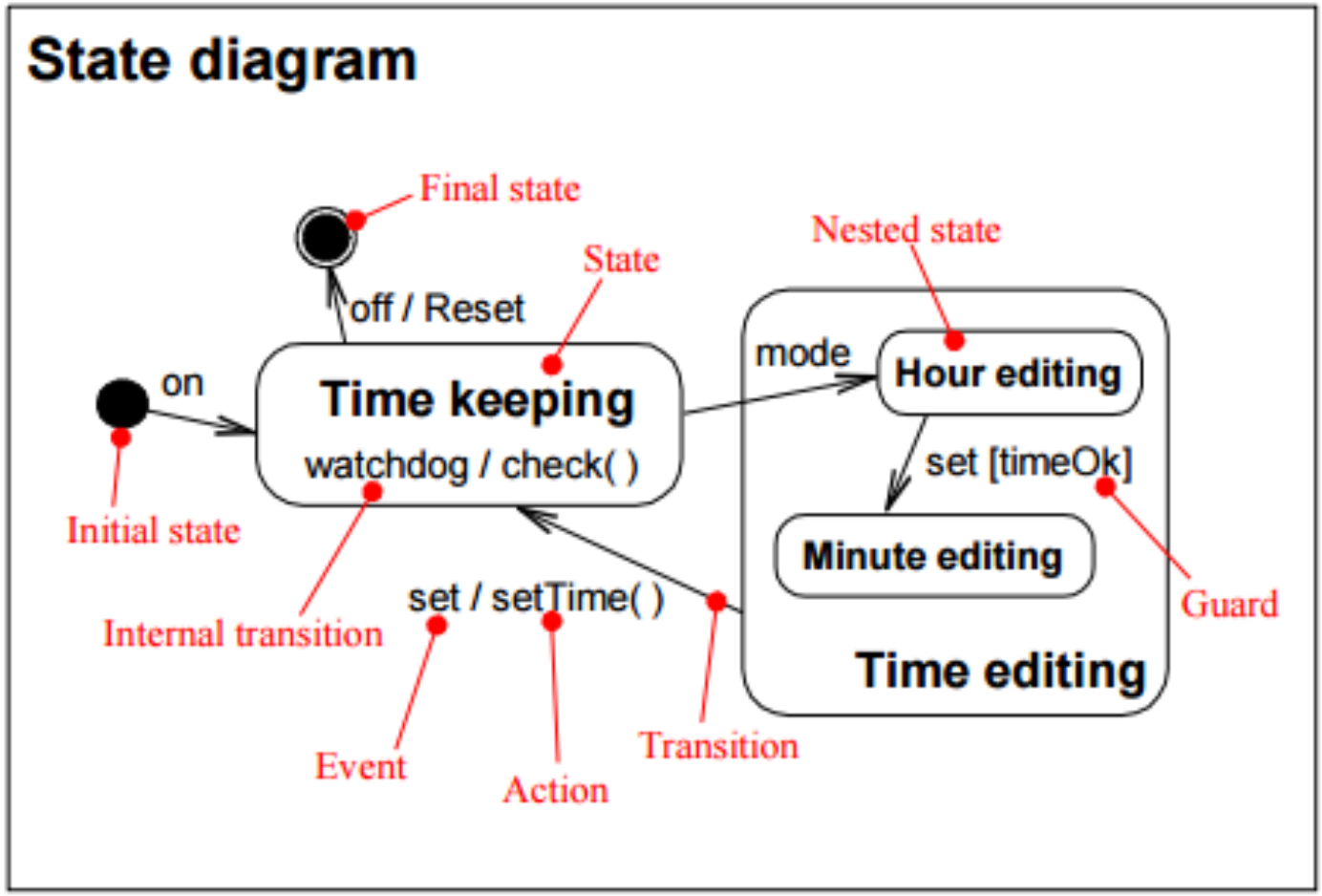
UML – Class Diagram



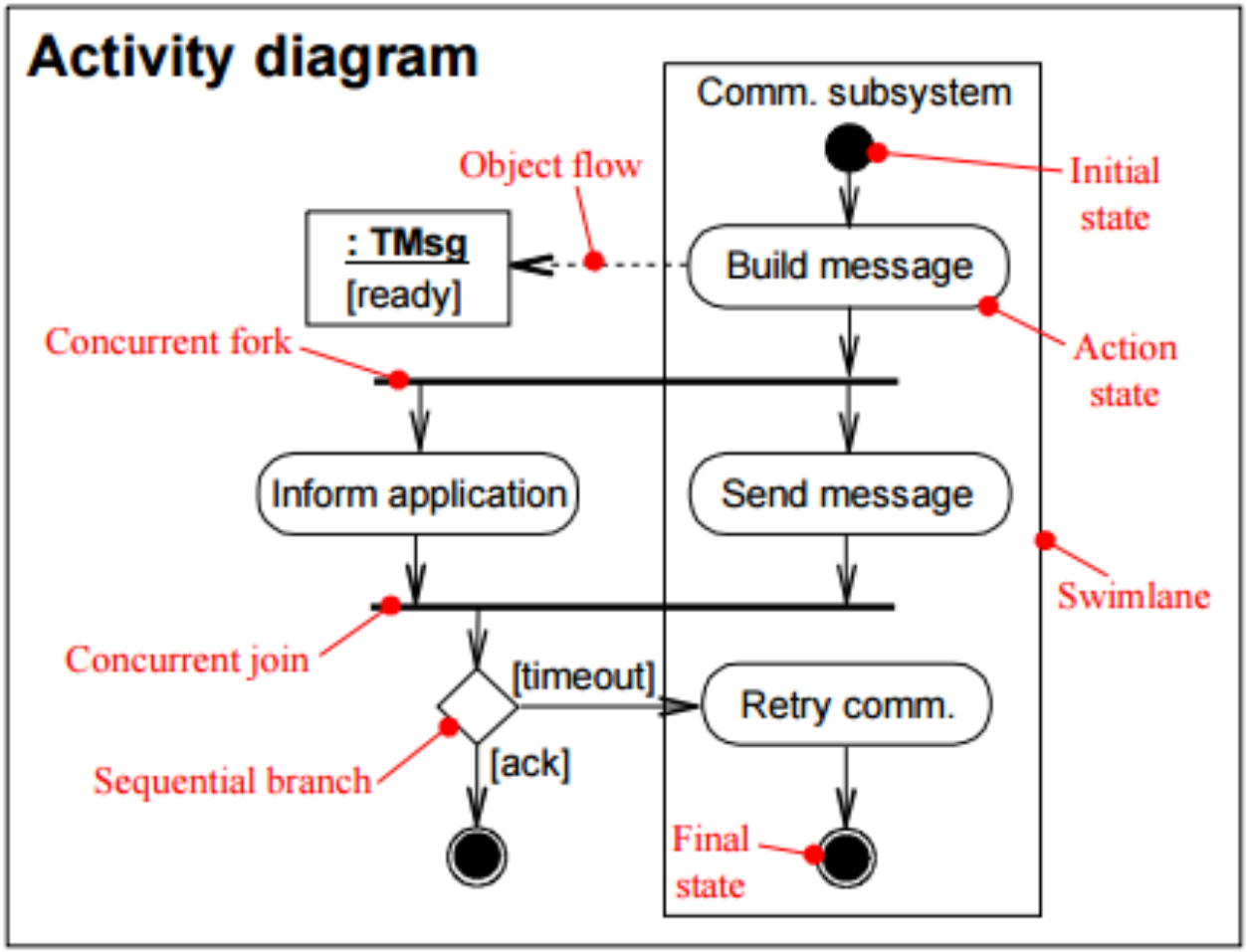
UML – Sequence Diagram



UML – State Diagram



UML – Activity Diagram



Design Patterns

- A *(software) design pattern* is a general and reusable solution to a commonly occurring problem within a given context in *software design*
  - It is not a finished design that can be transformed directly into source code
  - It is a description or a template for how to solve a problem that can be used in several different situations
- Design patterns are formalized *best practices* that the designer can use to solve common problems when designing a system
- *Object-oriented design patterns* typically show relationships and interactions among classes/objects, without specifying the final application classes/objects that are involved

GoF Design Patterns (I)

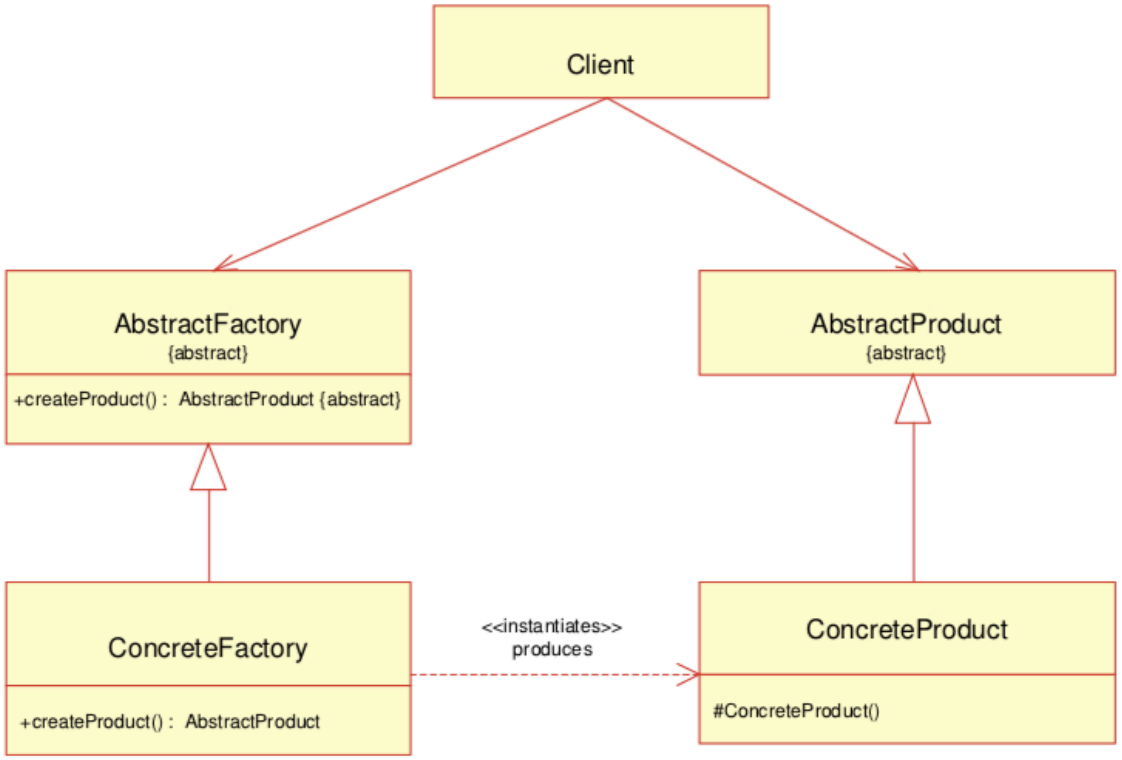
- Design patterns gained popularity in software engineering after that the book *Design Patterns: Elements of Reusable Object-Oriented Software* was published in 1994 by the so-called *Gang of Four (GoF)*, namely, E. Gamma, R. Helm, R. Johnson, J. Vlissides
- The GoF book discussed several design patterns categorized in the following classes
  - *Creational patterns*, which are intended to create objects, rather than having the programmer to instantiate objects directly
  - *Structural patterns*, which use inheritance to compose interfaces and define ways to compose objects to obtain new functionality
  - *Behavioral patterns*, which are specifically concerned with protocols for the communication among objects

Creational Patterns

- Creational patterns
  - *Abstract factory* groups object factories that have a common theme
  - *Builder* constructs complex objects by separating construction and representation
  - *Factory method* creates objects without specifying the exact class to create
  - *Prototype* creates objects by cloning an existing object
  - *Singleton* restricts object creation for a class to only one instance

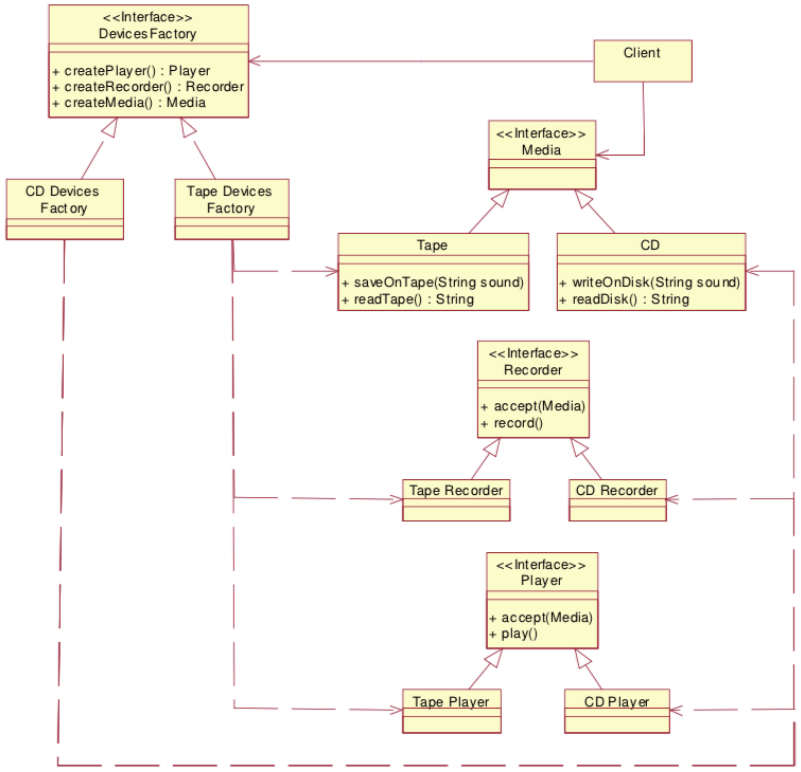
Abstract Factory (I)

- *Abstract factory* groups object factories that have a common theme



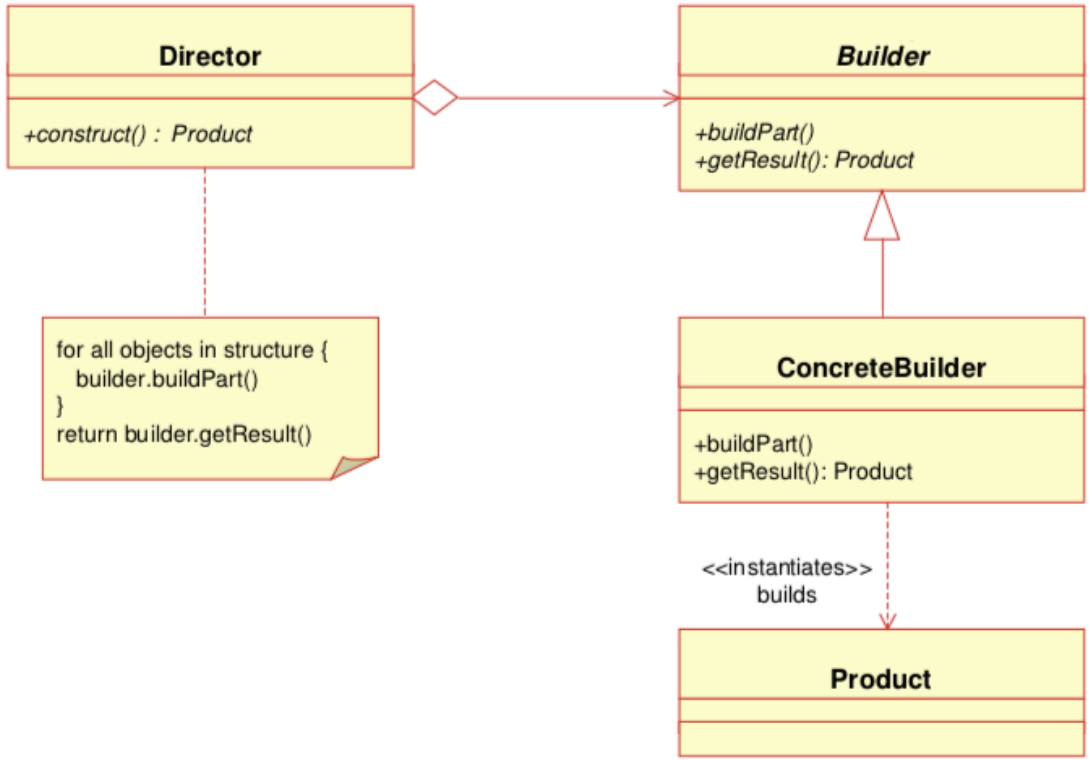
Abstract Factory (II)

- *Abstract factory* groups object factories that have a common theme



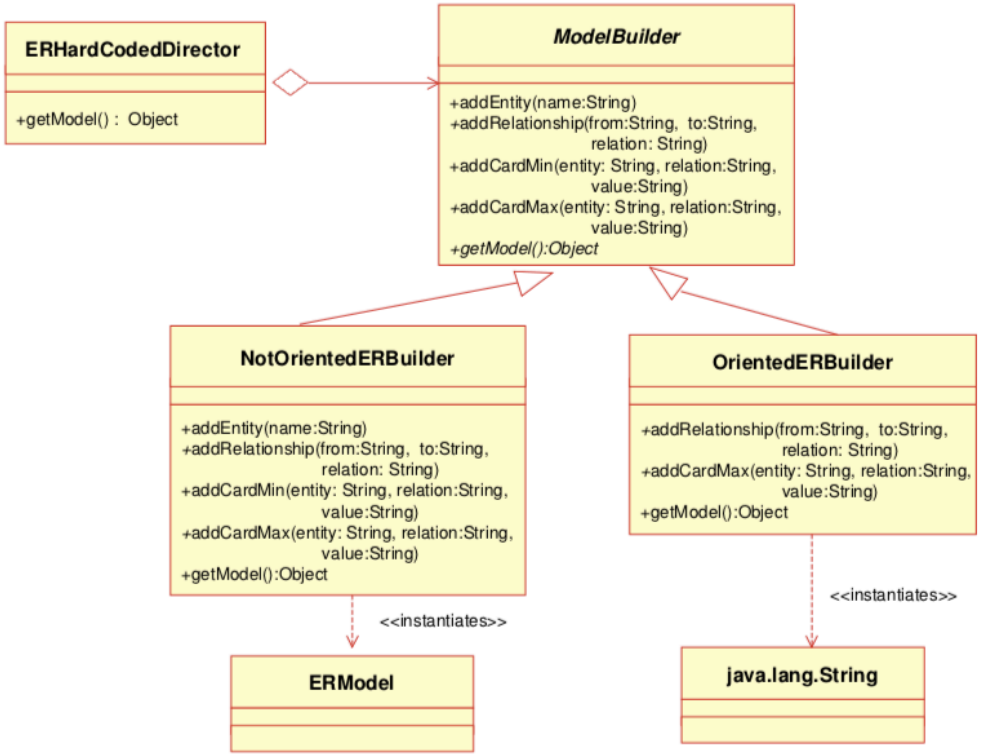
Builder (I)

- *Builder* constructs complex objects by separating construction and representation



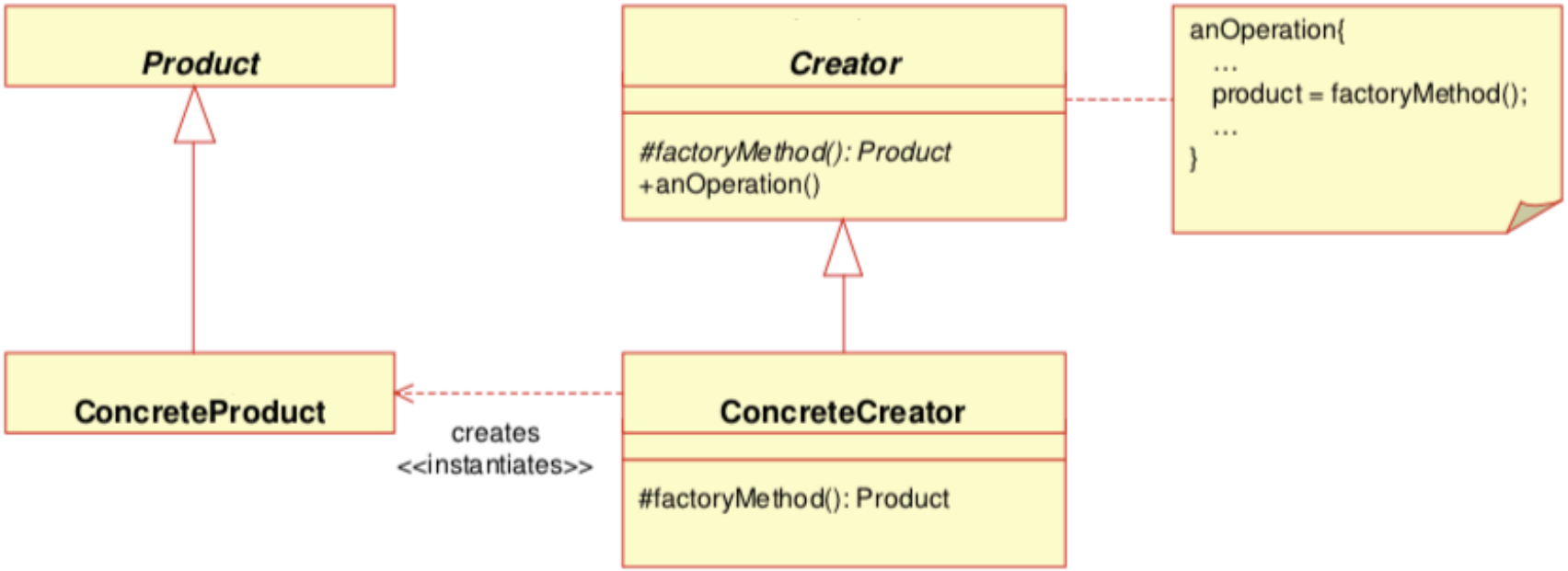
Builder (II)

- *Builder* constructs complex objects by separating construction and representation



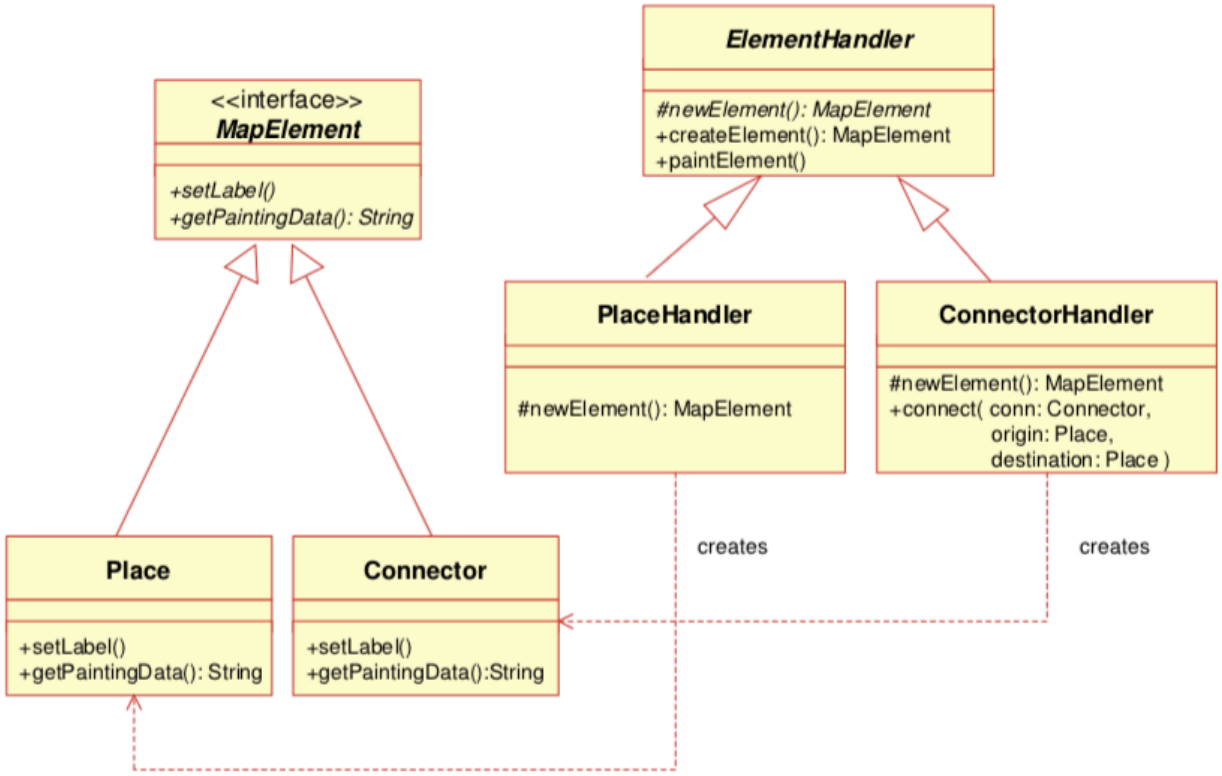
Factory Method (I)

- *Factory method* creates objects without specifying the exact class to create



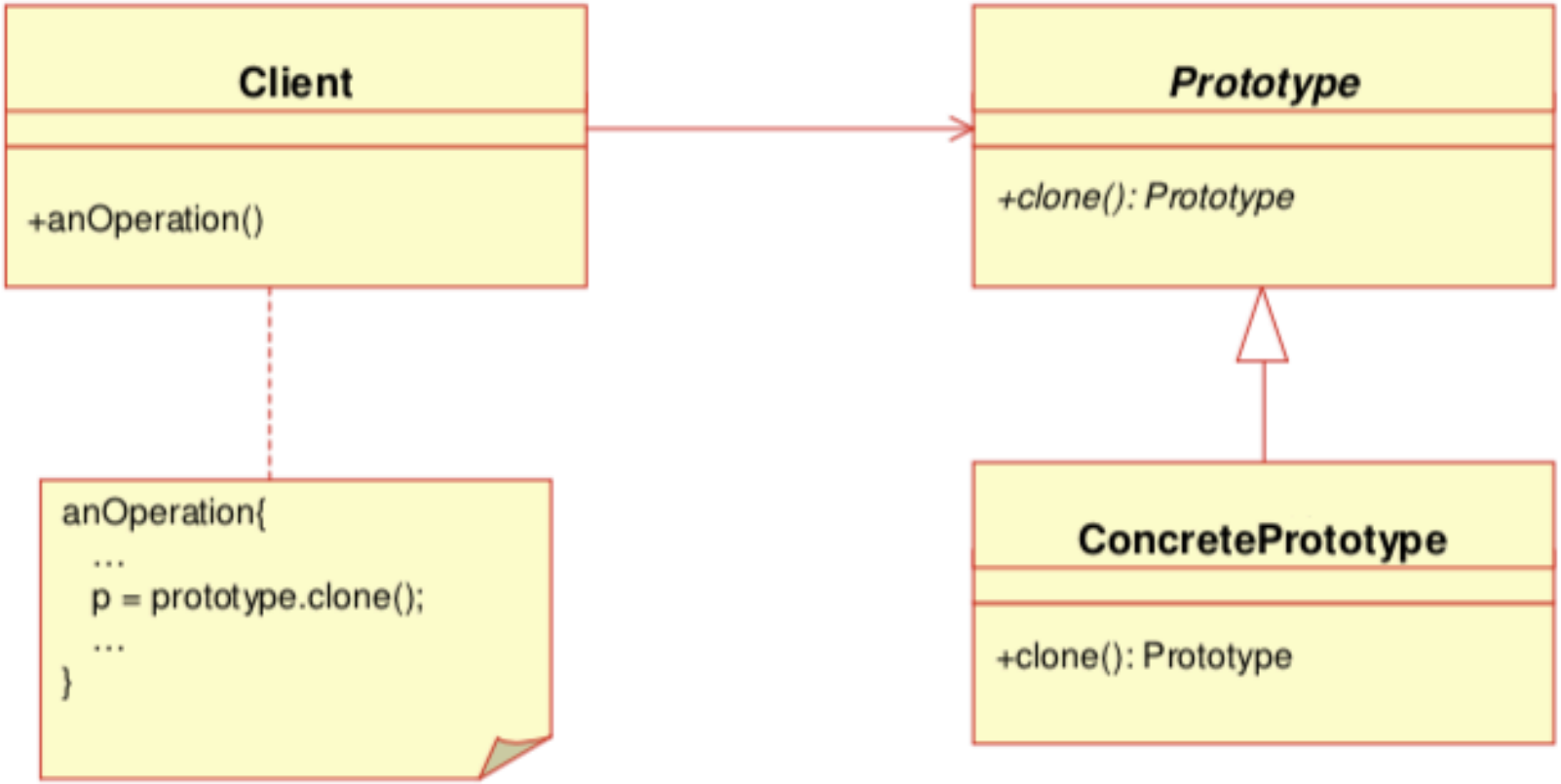
Factory Method (II)

- *Factory method* creates objects without specifying the exact class to create



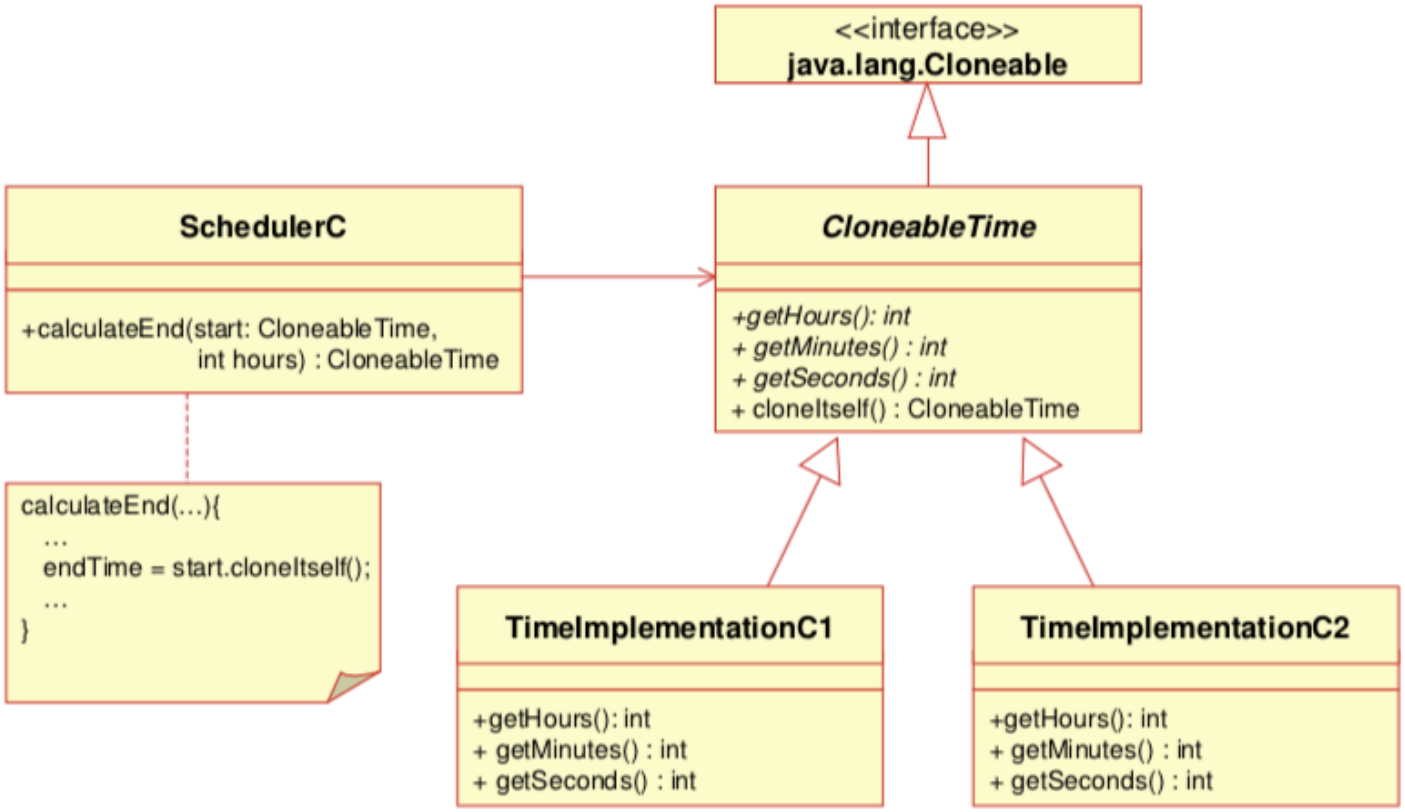
Prototype (I)

- *Prototype* creates objects by cloning an existing object



Prototype (II)

- *Prototype* creates objects by cloning an existing object



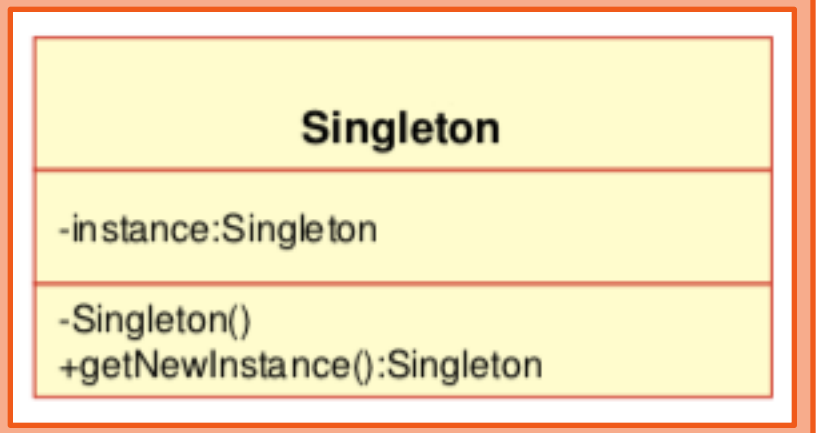
Singleton

```
public class SingletonClass {
    private static volatile SingletonClass instance;

    public static SingletonClass getInstance() {
        if (instance == null) {
            synchronized (SingletonClass.class) {
                if (instance == null) {
                    instance = new SingletonClass();
                }
            }
        }

        return instance;
    }

    private SingletonClass() {}
}
```



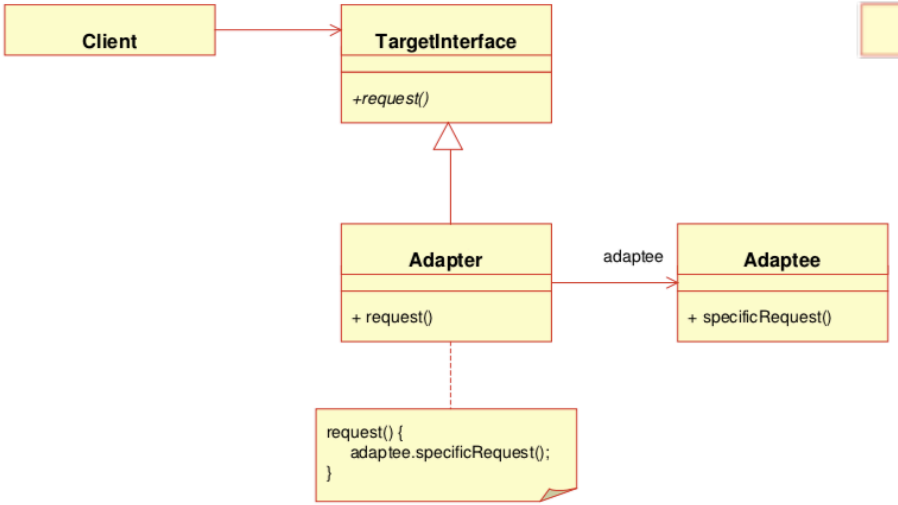
Structural Patterns

- Structural patterns
  - *Adapter* allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class
  - *Bridge* decouples an abstraction from its implementation so that the two can vary independently
  - *Composite* composes zero-or-more similar objects so that they can be manipulated as one object
  - *Decorator* dynamically adds/overrides behavior in an existing method of an object
  - *Proxy* provides a placeholder for another object to control access, and reduce complexity

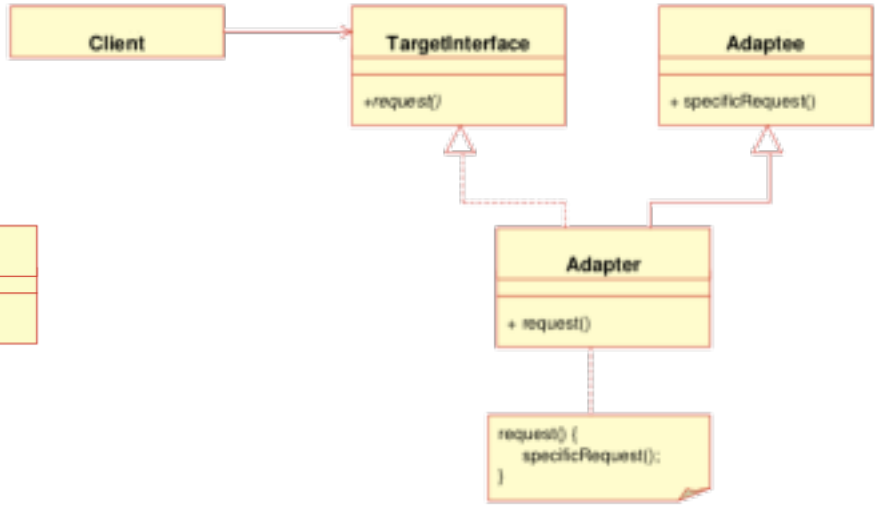
Adapter (I)

- *Adapter* allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class

Object adapter



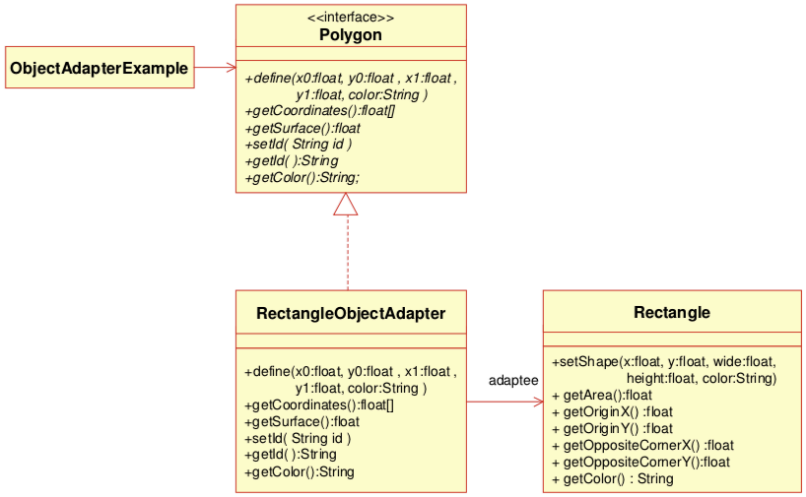
Class adapter



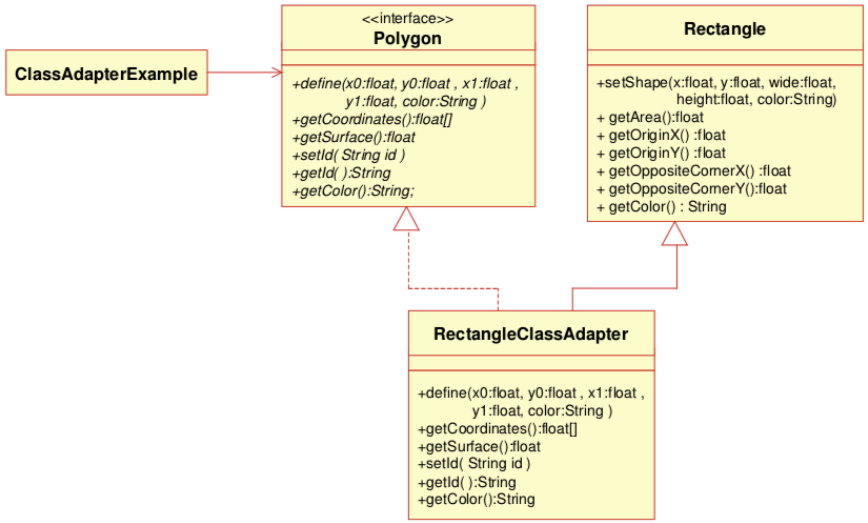
Adapter (II)

- *Adapter* allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class

Object adapter

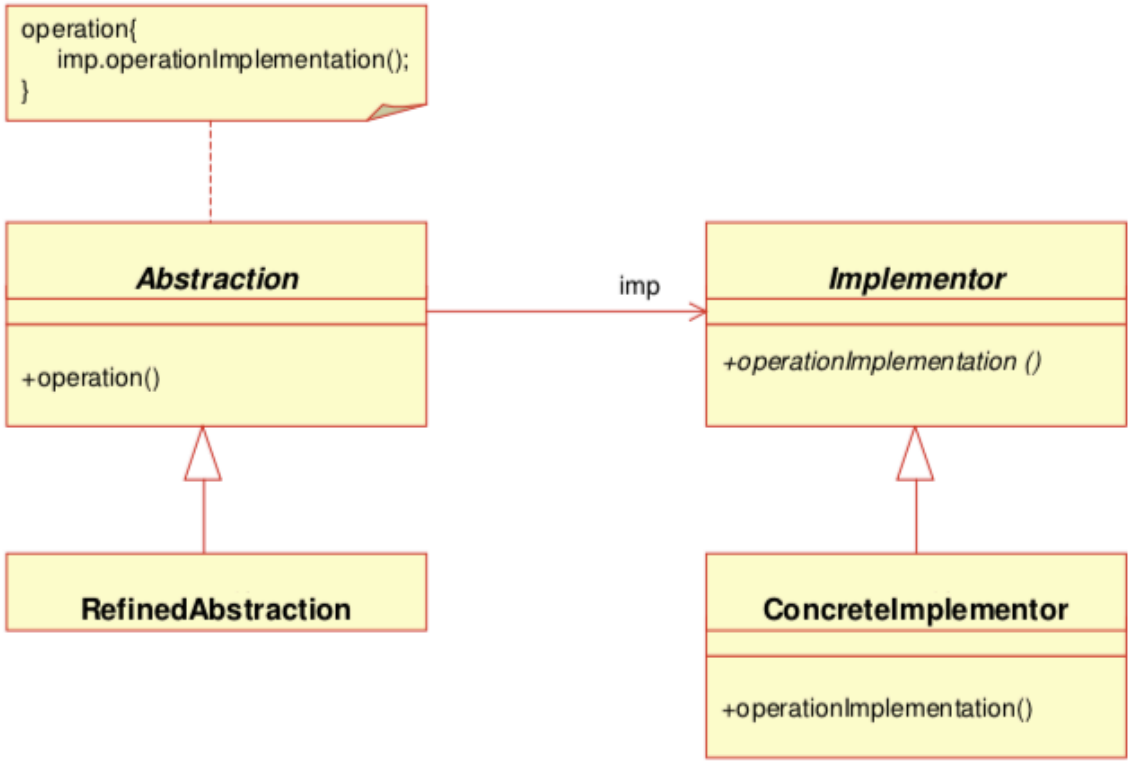


Class adapter



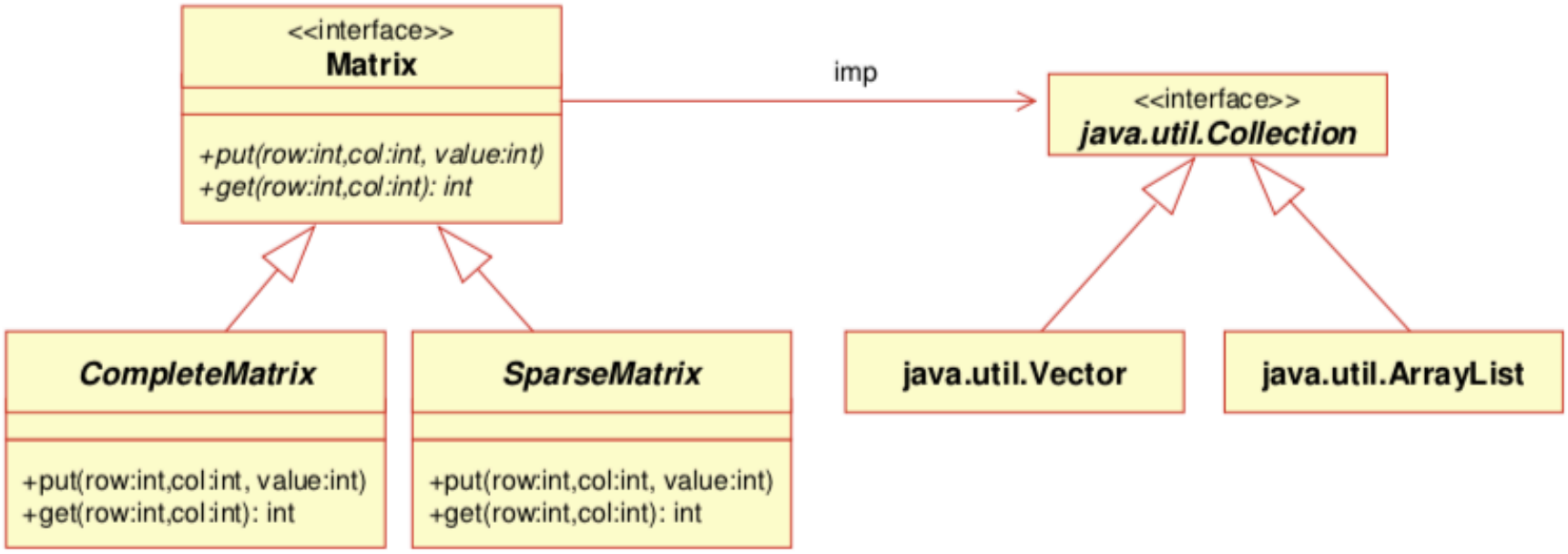
Bridge (I)

- *Bridge* decouples an abstraction from its implementation so that the two can vary independently



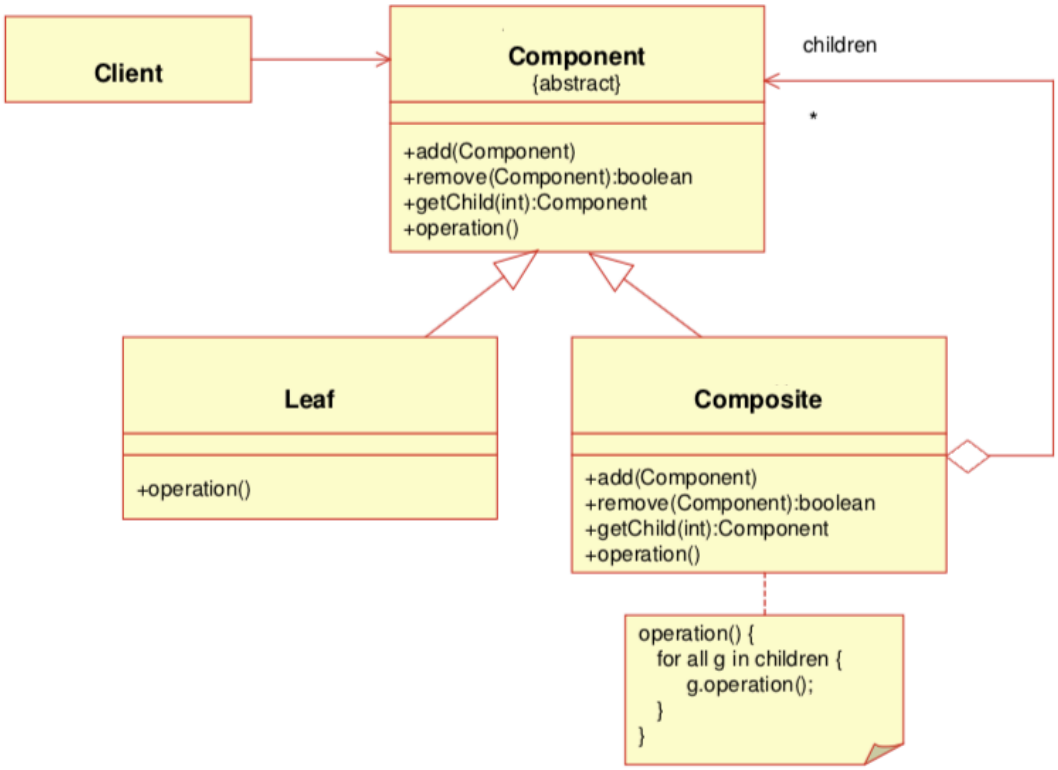
Bridge (II)

- *Bridge* decouples an abstraction from its implementation so that the two can vary independently



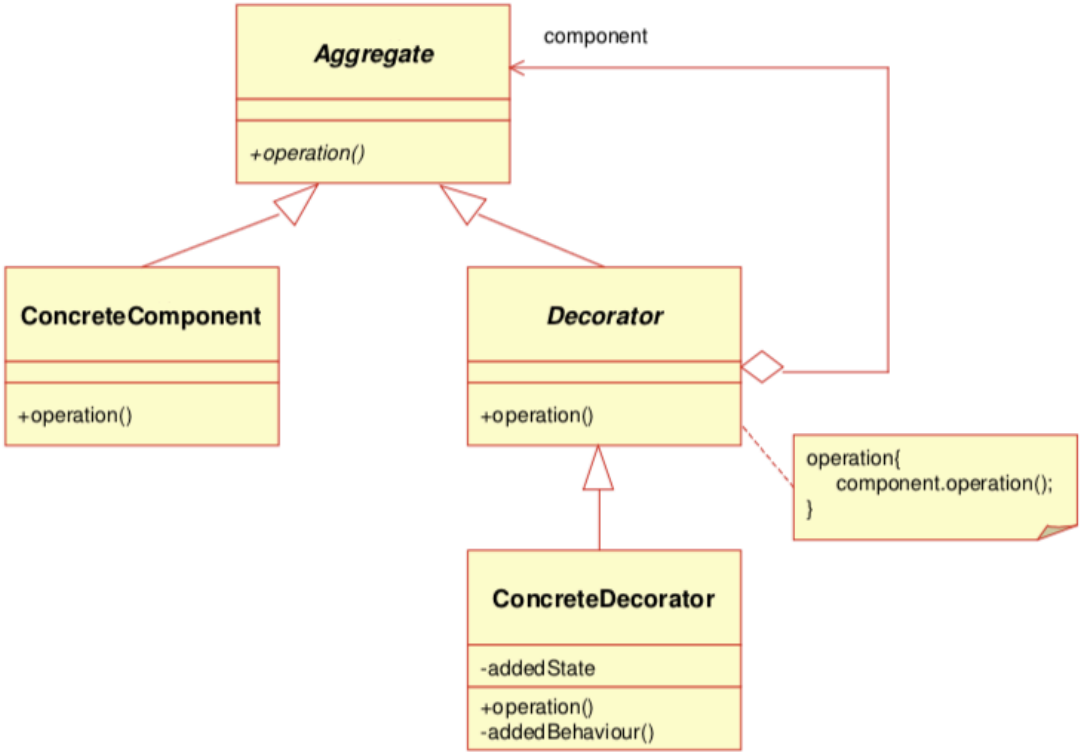
Composite

- *Composite* composes zero-or-more similar objects so that they can be manipulated as one object



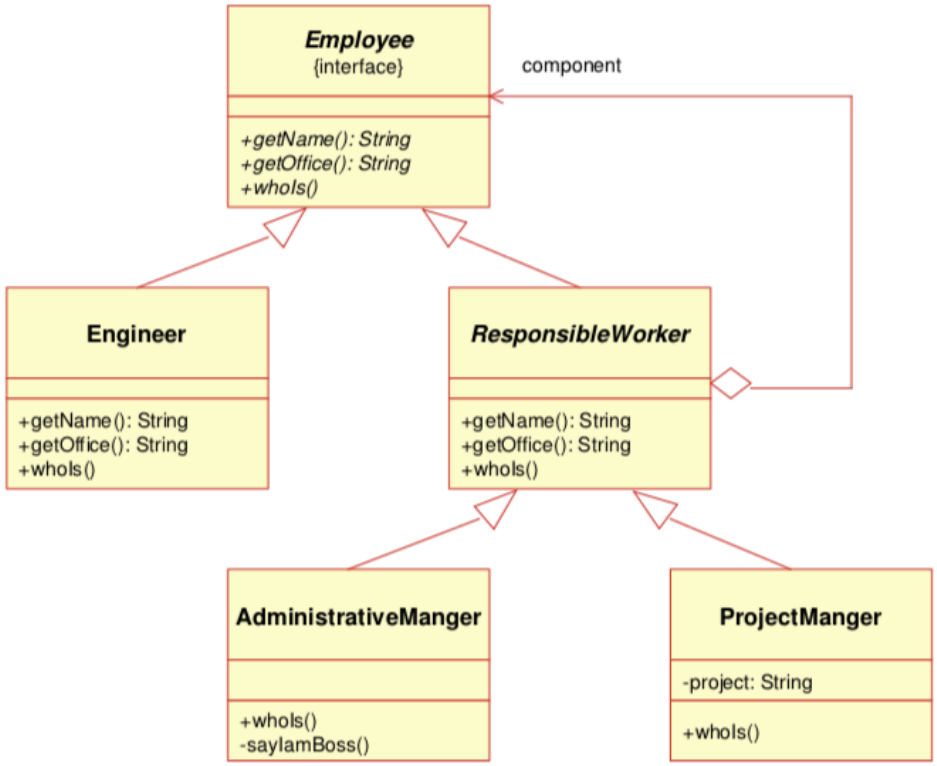
Decorator (I)

- *Decorator* dynamically adds/overrides behavior in an existing method of an object



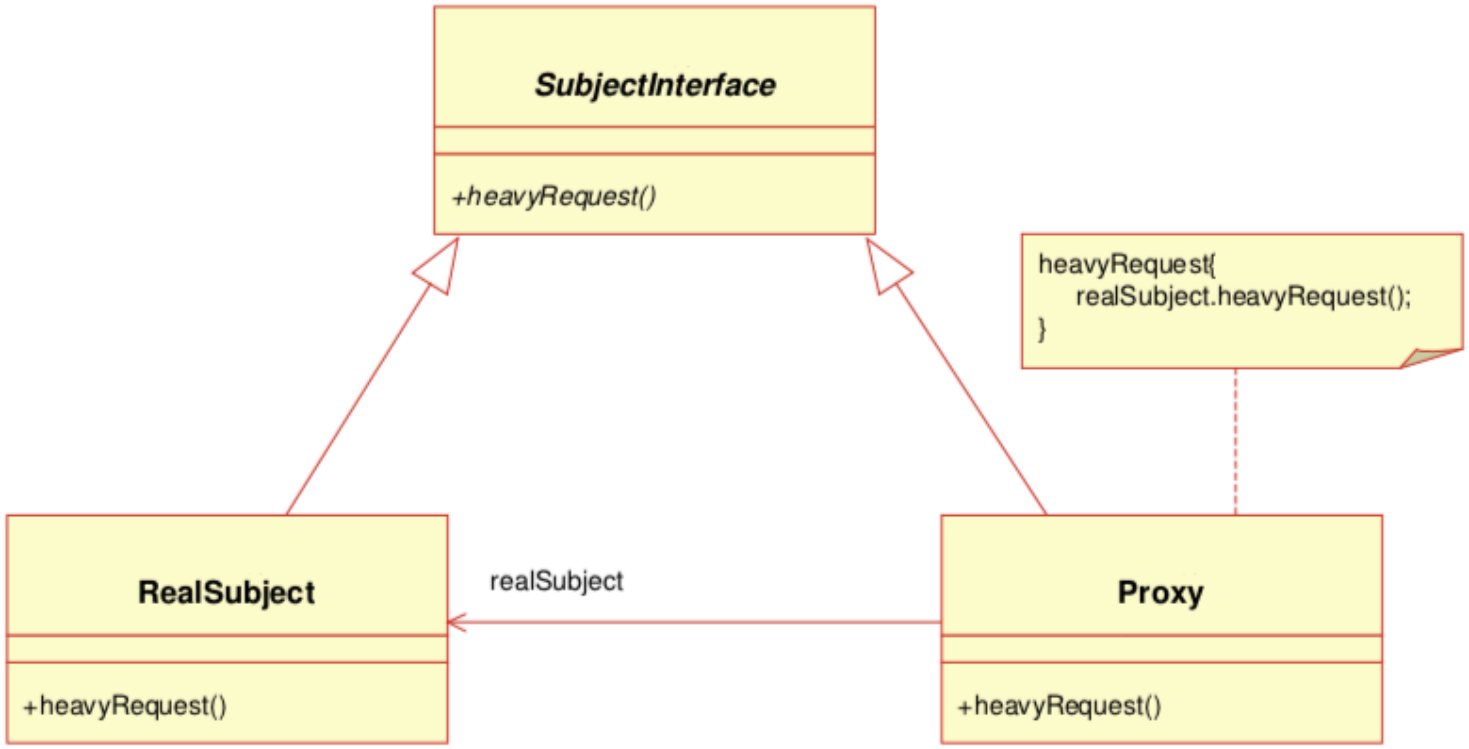
Decorator (II)

- *Decorator* dynamically adds/overrides behavior in an existing method of an object



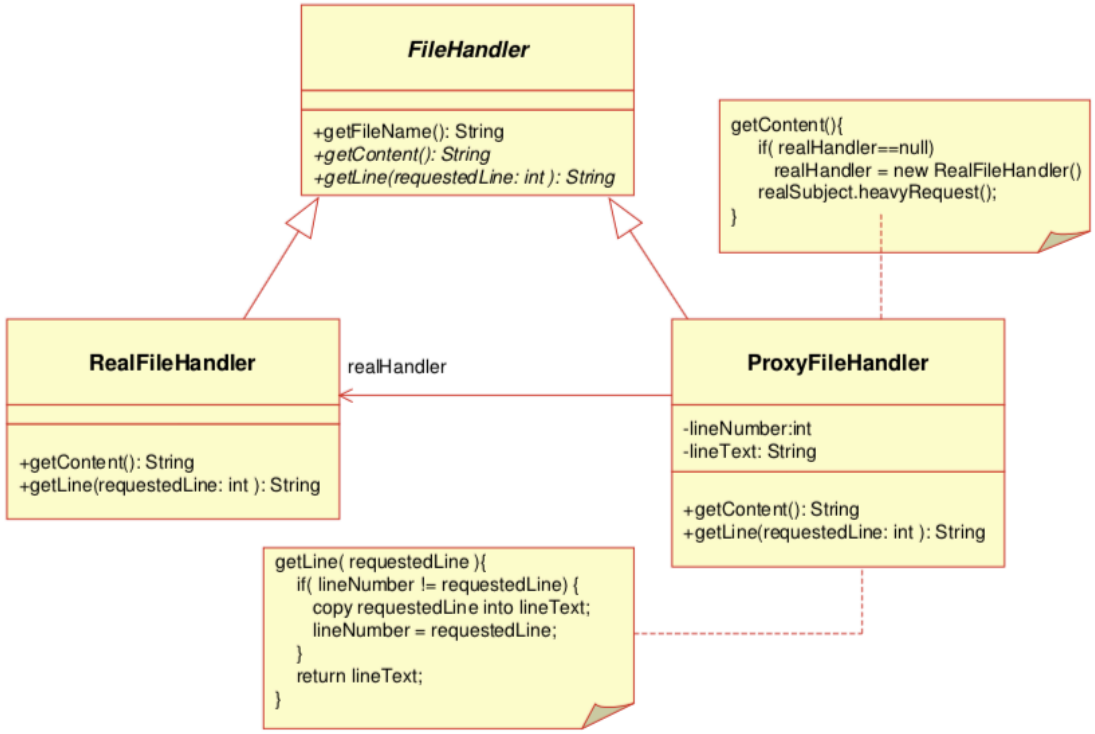
Proxy (I)

- *Proxy* provides a placeholder for another object to control access, and reduce complexity



Proxy (II)

- *Proxy* provides a placeholder for another object to control access, and reduce complexity

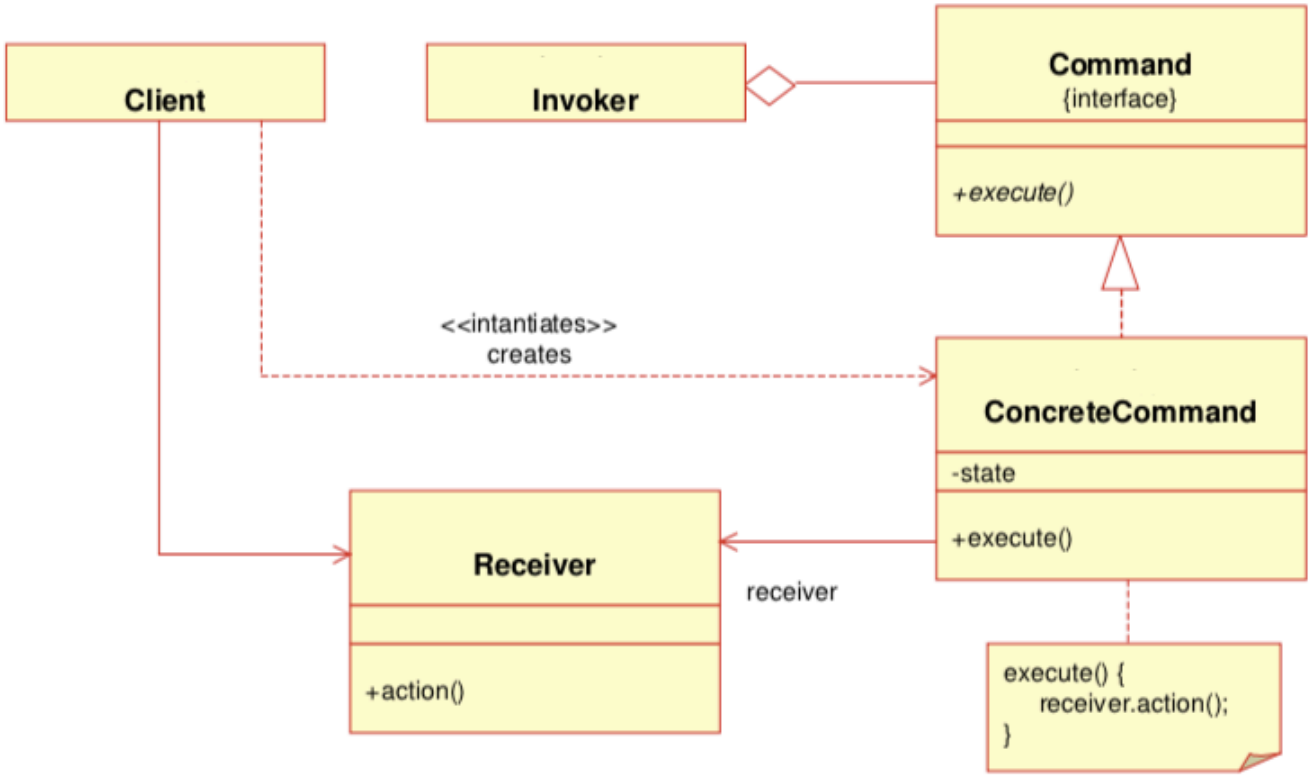


Behavioral Patterns

- Behavioral patterns
  - *Command* creates objects that encapsulate actions and parameters
  - *Interpreter* implements an interpreter for a specialized language, which is often a *DSL (Domain-Specific Language)*
  - *Iterator* accesses the elements of an object sequentially without exposing its underlying representation
  - *Observer* is a publish/subscribe pattern, which allows a number of observer objects to see an event
  - *Visitor* separates an algorithm from an object structure by moving the hierarchy of methods into one object

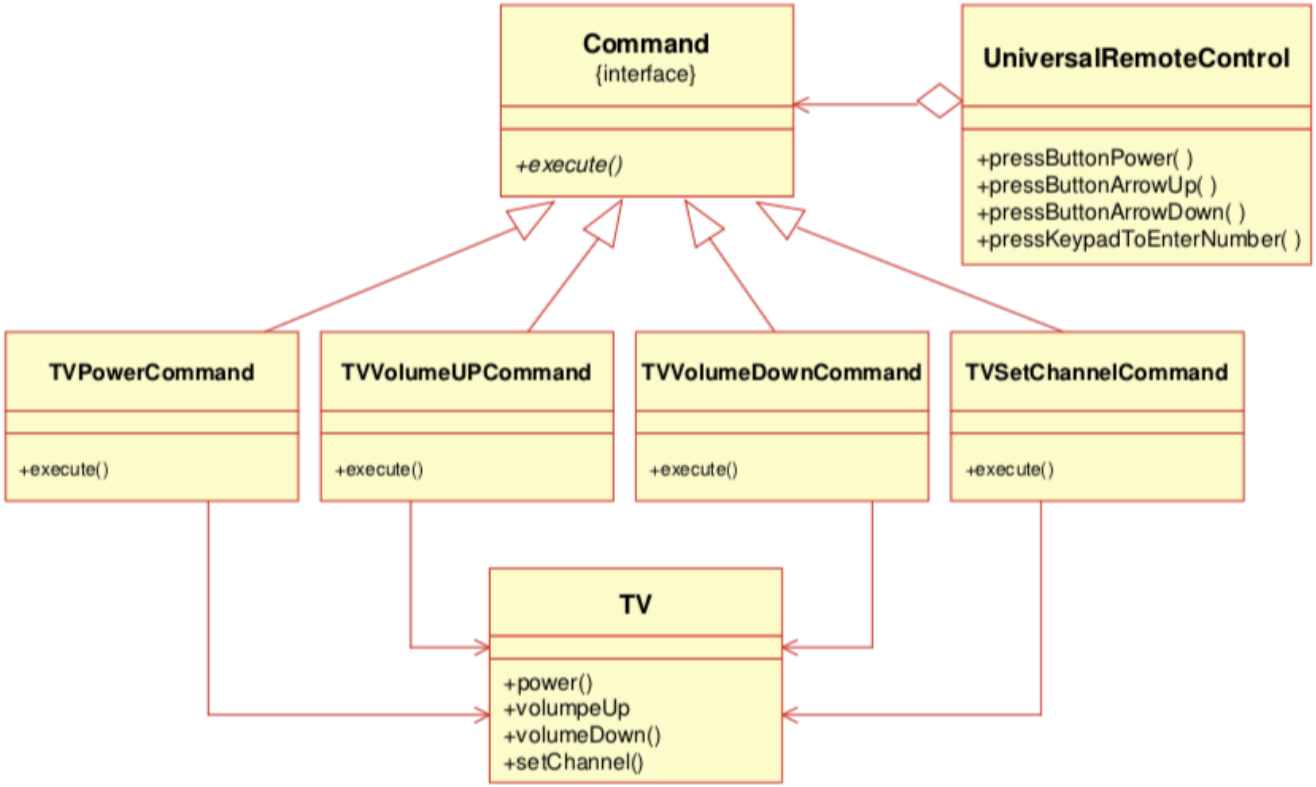
Command (I)

- *Command* creates objects that encapsulate actions and parameters



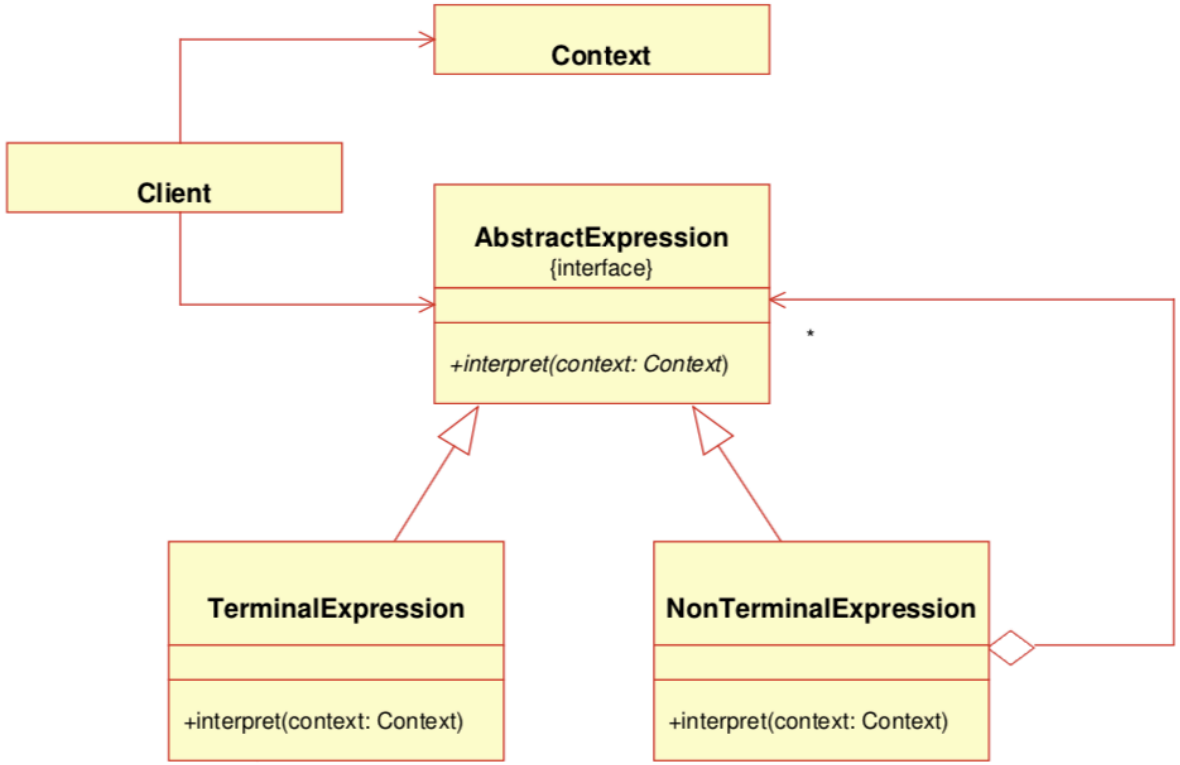
Command (II)

- *Command* creates objects that encapsulate actions and parameters



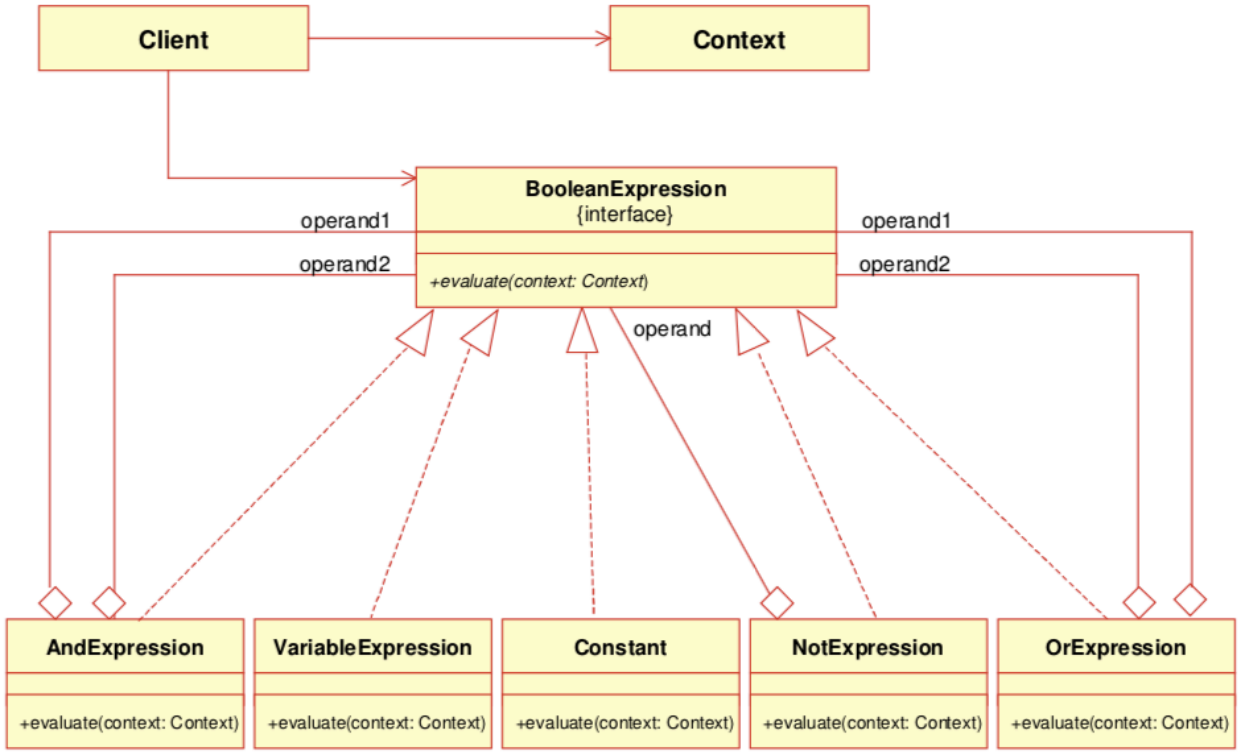
Interpreter (I)

- *Interpreter* implements an interpreter for a specialized language, which is often a DSL



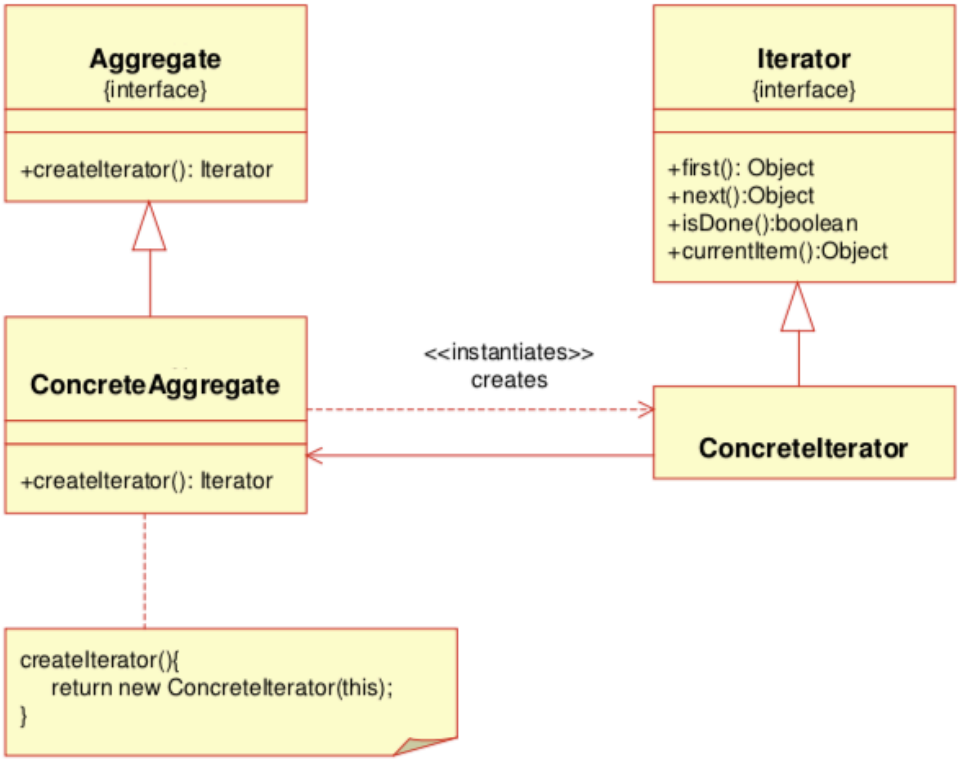
Interpreter (II)

- *Interpreter* implements an interpreter for a specialized language, which is often a DSL



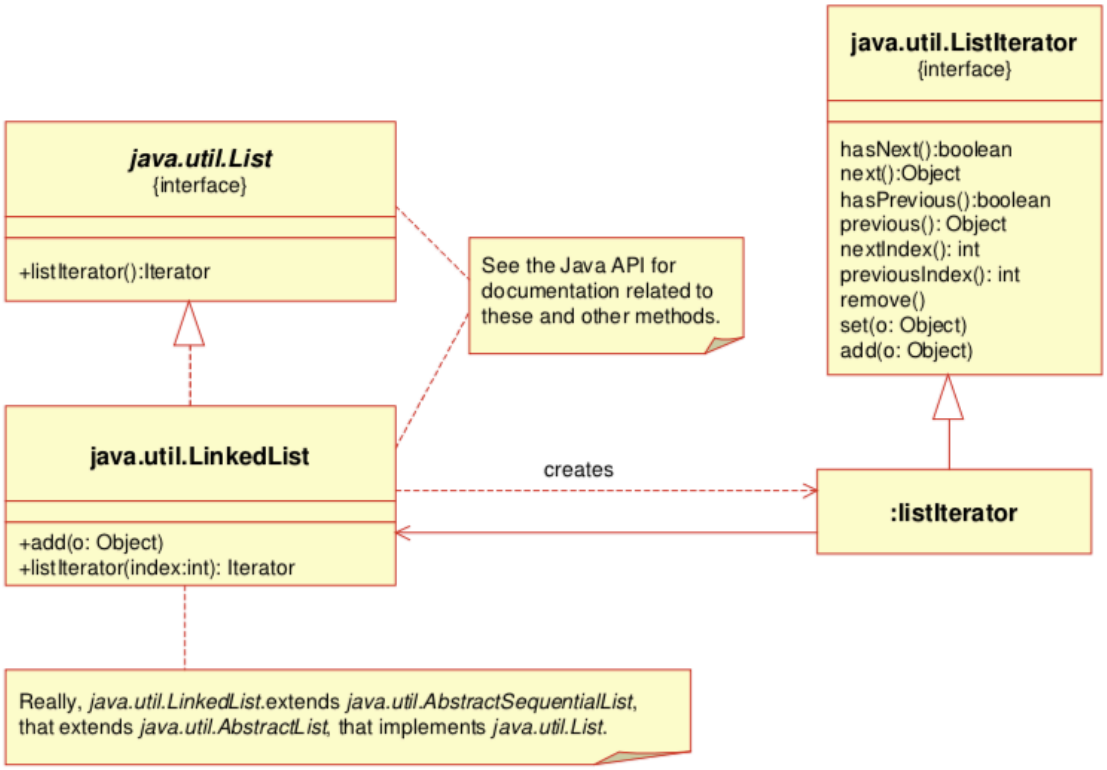
Iterator (I)

- *Iterator* accesses the elements of an object sequentially without exposing its underlying representation



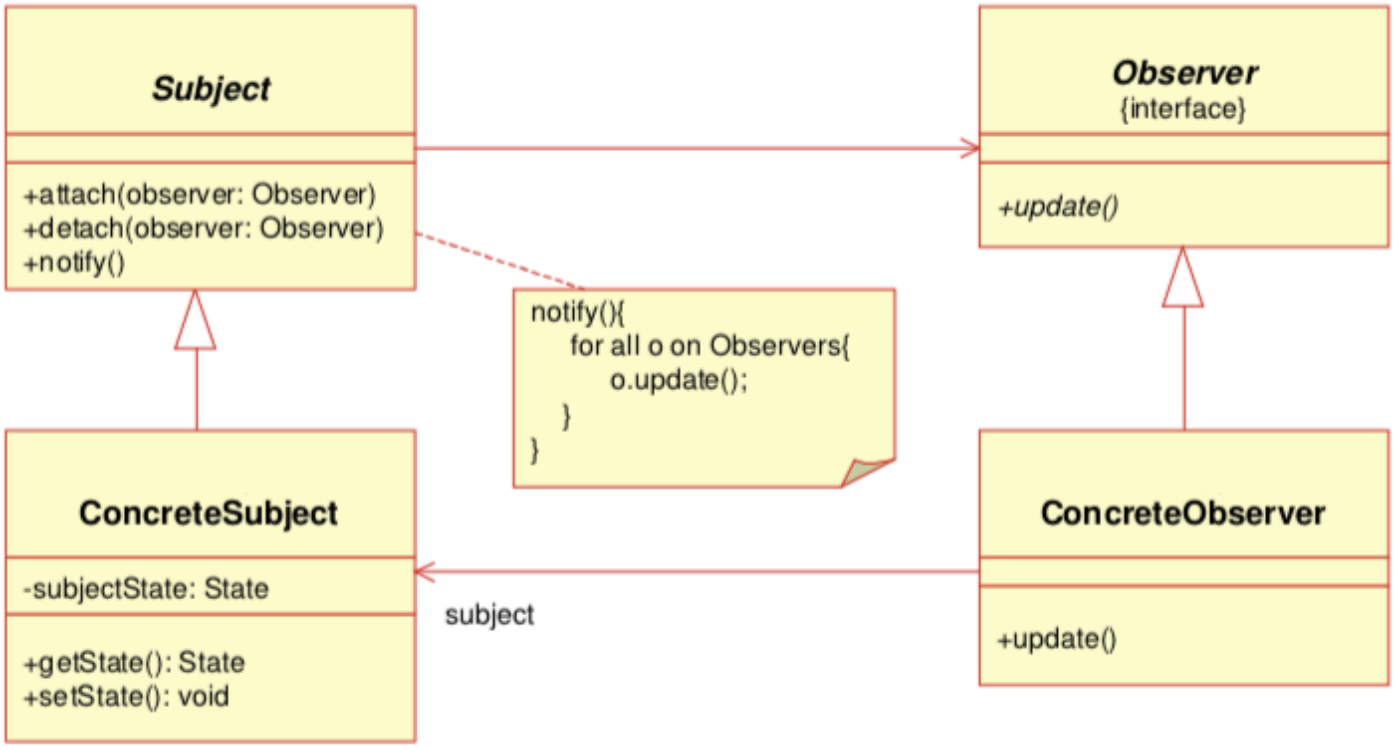
Iterator (II)

- *Iterator* accesses the elements of an object sequentially without exposing its underlying representation



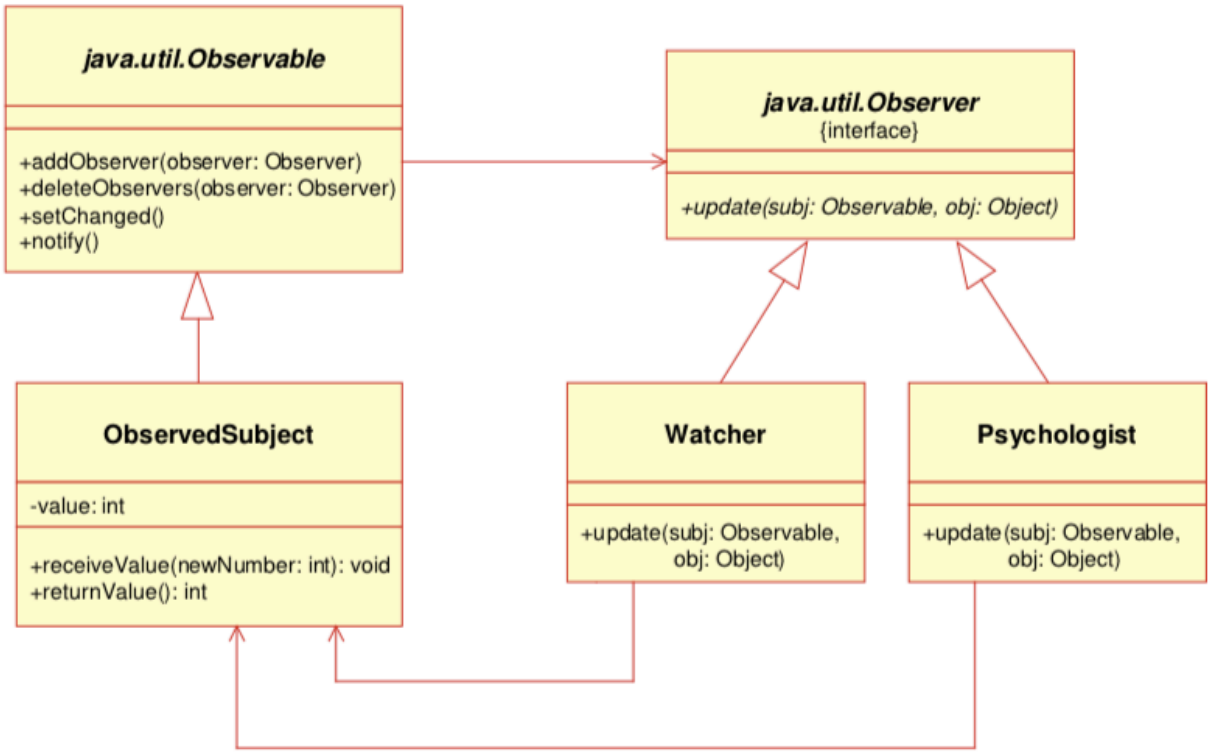
Observer (I)

- *Observer* is a publish/subscribe pattern, which allows a number of observer objects to see an event



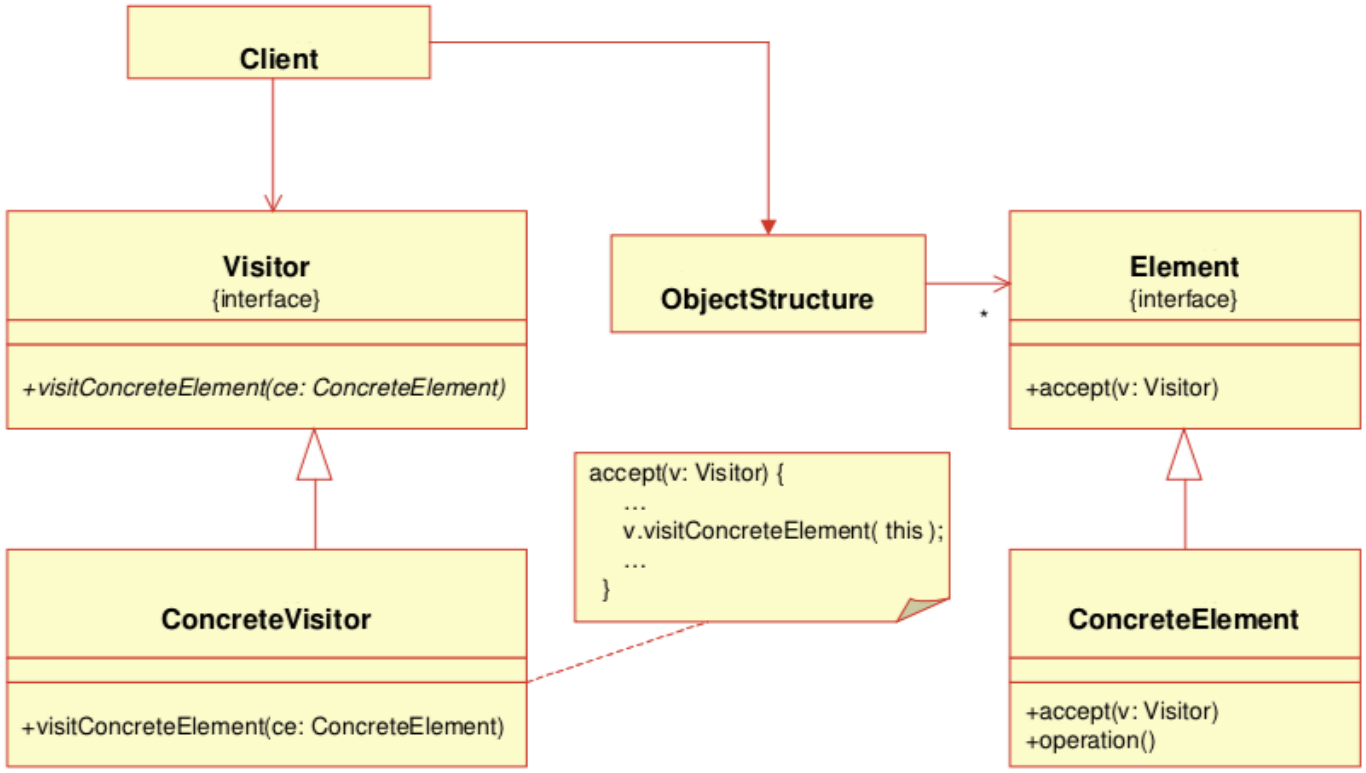
Observer (II)

- *Observer* is a publish/subscribe pattern, which allows a number of observer objects to see an event



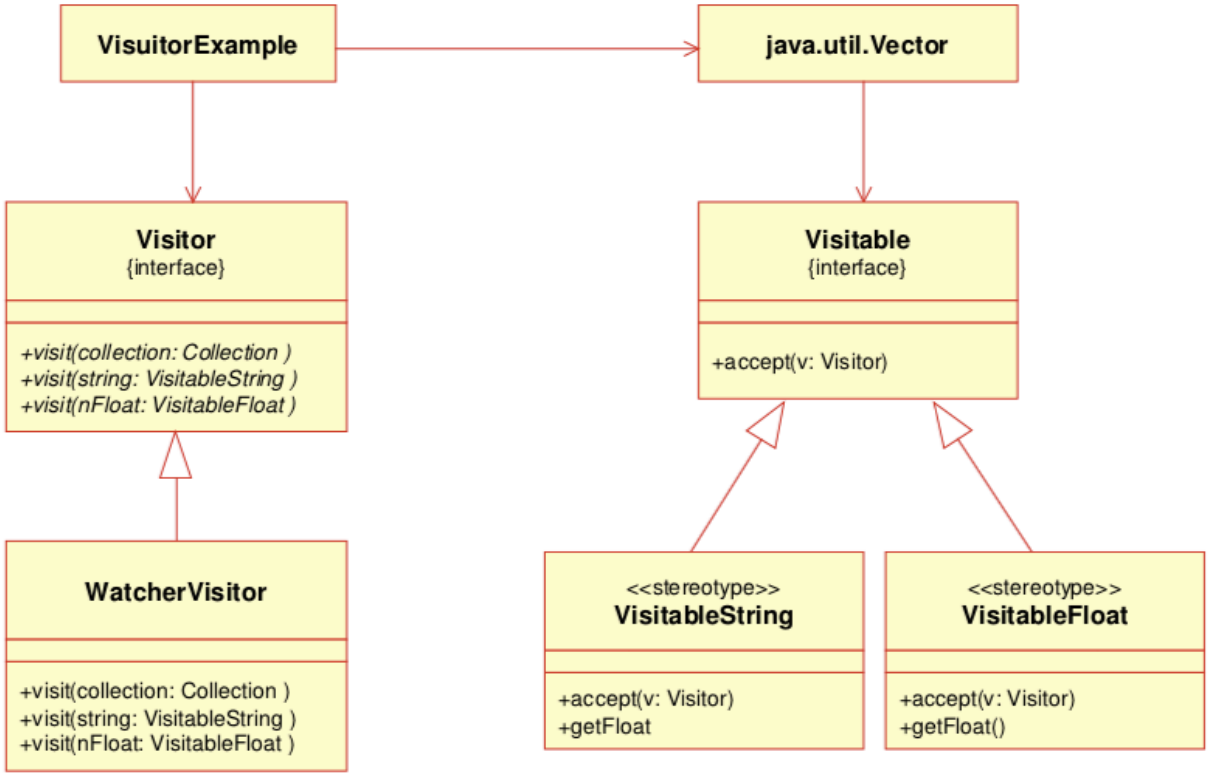
Visitor (I)

- *Visitor* separates an algorithm from an object structure by moving the hierarchy of methods into one object



Visitor (II)

- *Visitor* separates an algorithm from an object structure by moving the hierarchy of methods into one object



Software Engineering

Prof. Federico Bergenti
Artificial Intelligence Laboratory
www.ailab.unipr.it/bergenti

www.ailab.unipr.it

Artificial Intelligence Laboratory
Università degli Studi di Parma
Parco Area delle Scienze 53/A
43124 Parma (Italy)

ailab@unipr.it