Software Engineering

Elegant tools for a more civilized age.

Dr. Hale
University of Nebraska at Omaha
SDLC, Requirements, and Use Cases

Today's Class

Part 1: A Quick look at the development space and SDLC

- Different development options available
- Picking the right tool for the job
- Asking the right questions
- Software Development Risks

Break

Today's Class

Part 2: Software Requirements, Use cases, and Misuse cases

- Defining and communicating requirements
- Who are the (good/bad) actors?
- What are their goals?
- How does it affect your application?

Hands-on

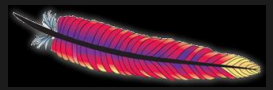
- Create a use-case/misuse case diagram

Part 1: A quick look at the project management tips and Software Development Lifecycle

So many options…

serverside

clientside



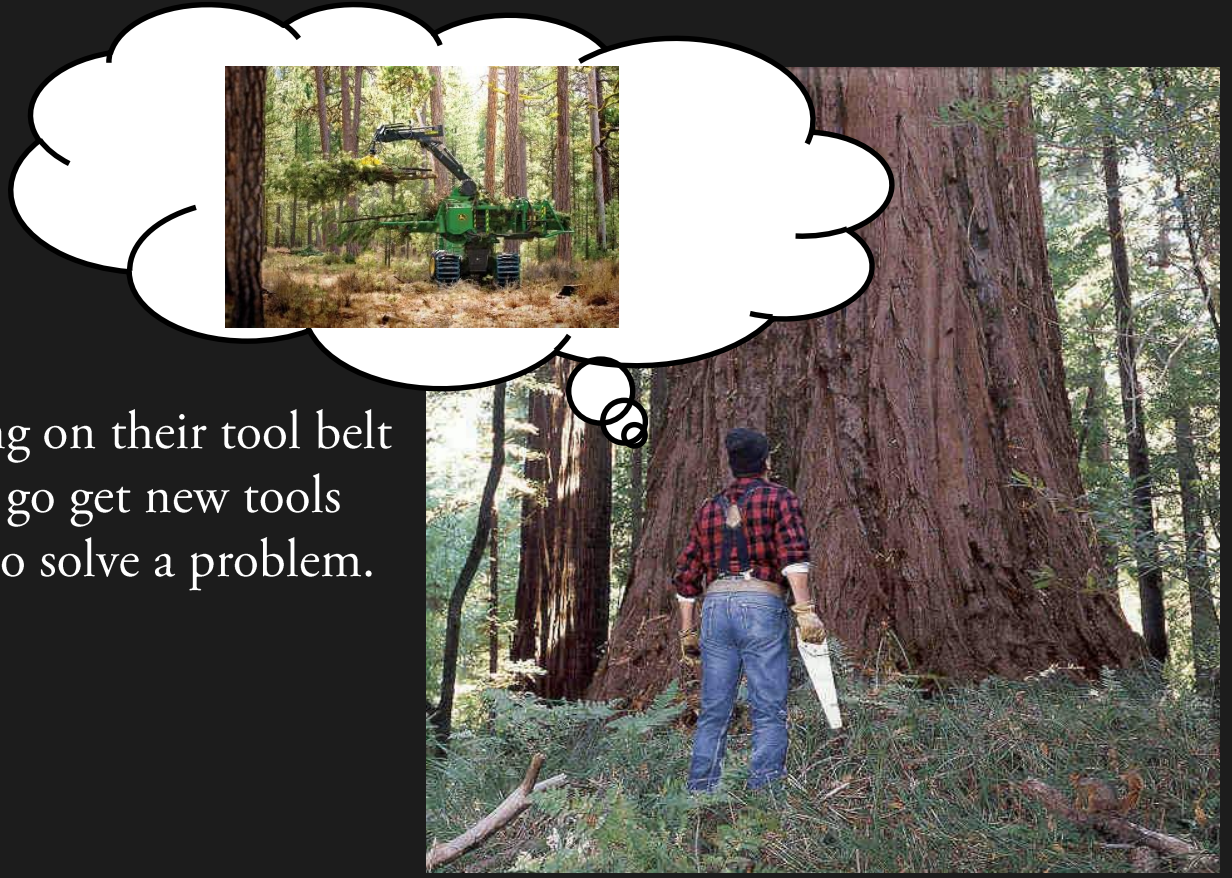
Developer tendencies

Bad Developers tend to use the same tool(s) to solve all problems, even if its less than ideal.

Developer tendencies

This can mean full or partial project failure.

Developer tendencies



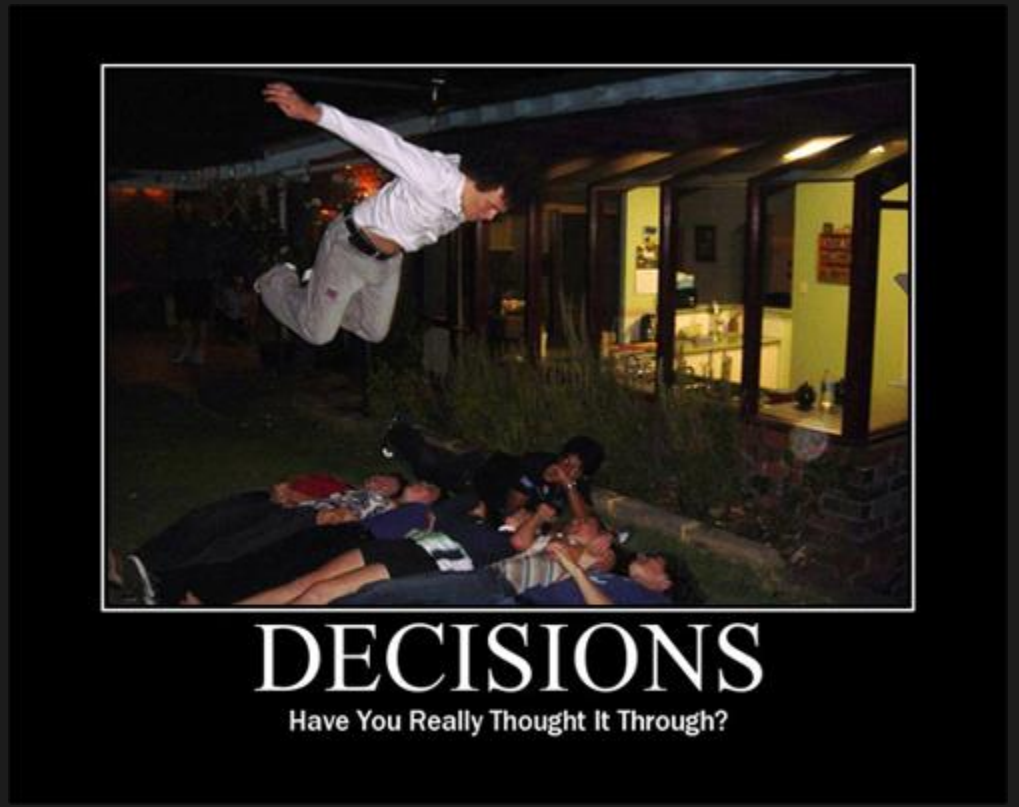
Good developers use everything on their tool belt or know when they need to go get new tools (or new specialized workers) to solve a problem.

Ask the right questions

What platforms/devices need to be supported?
What languages does your team know?
What are your application needs?
Does it need to be fast? secure? updated regularly? etc
What are the costs (money/time) and are they going to pay off?
How will the application be structured? (will return)
How will it use existing resources? (will return)
What are the project risks? (will return)

Ask the right questions

Your goal should be to help get the development team to think critically about their design decisions and back them up with hard facts that can pass the management cost/benefit analysis.

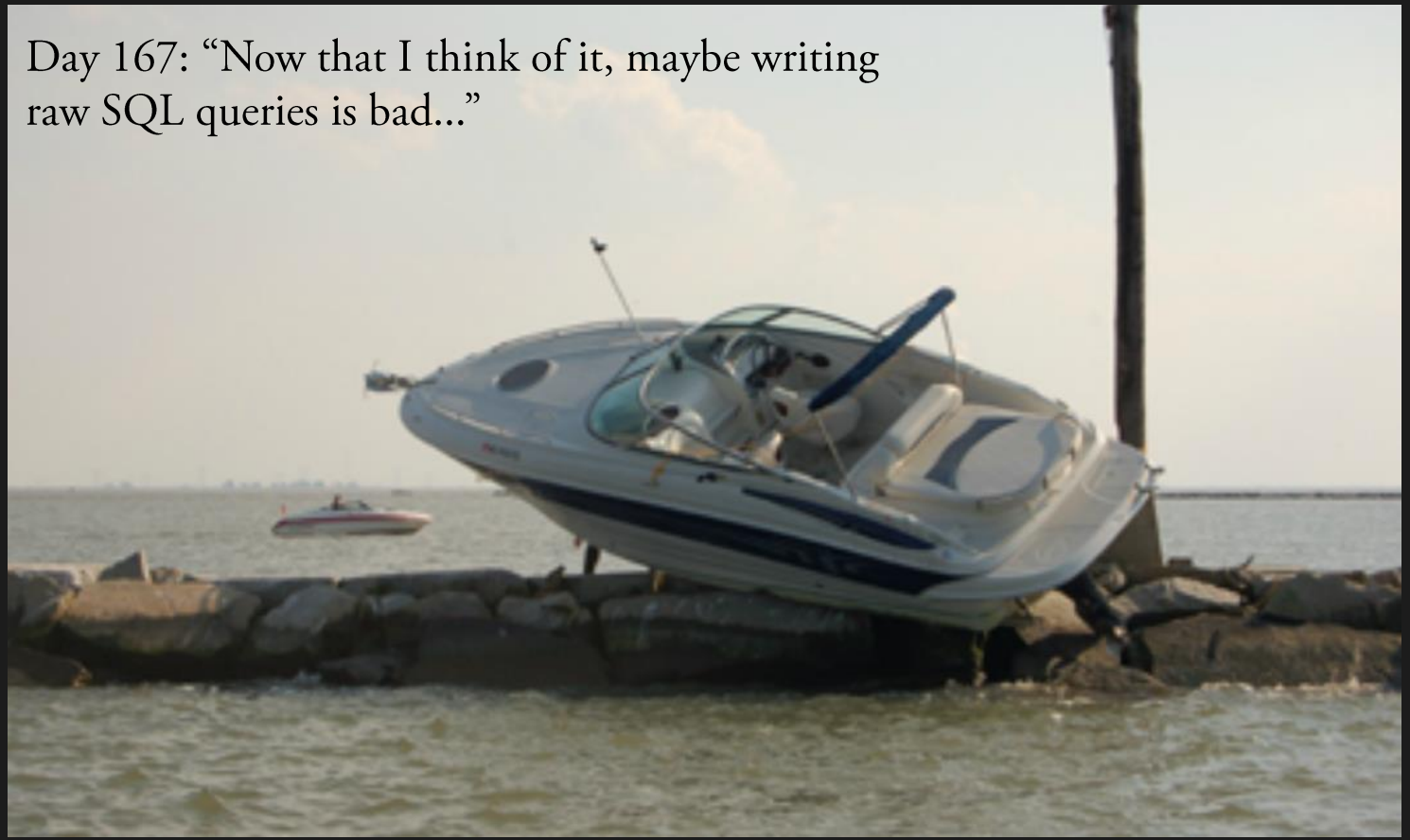


Software Development Risks

Project variables that endanger or eliminate the success of the project.

- Examples
  - Schedule delays
  - Poor initial design
  - Scalability issues
  - Team member inadequacies
  - Poor language / framework choices
  - Bad coding practices / documentation

Software Development Risks



Calculating SDLC Risks

Each risky project variable can be quantified in terms of the probability of loss and the potential impact of the loss. (just like security risk)

(probably) Familiar equation:
*Risk Exposure = Prob(loss) * Size(loss)*

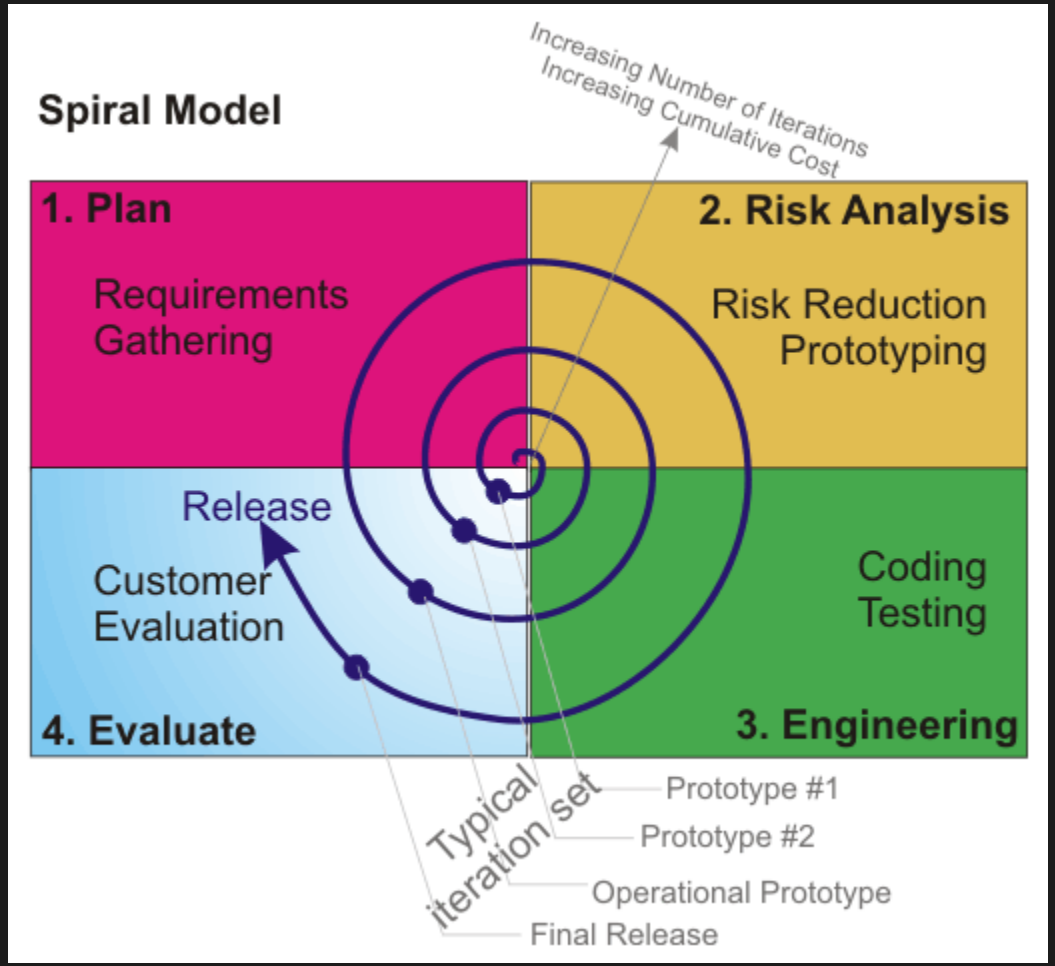
Prob(loss) can be reduced by assessing, monitoring, and/or mitigating risky project variables (e.g. pick a language familiar to the team, train the weaker team members, select good components/frameworks)

Size(loss) can be reduced significantly by reducing the impact of the risky variables. For design and coding this includes things like defect/code inspections, good unit testing practices, and modular architectural design

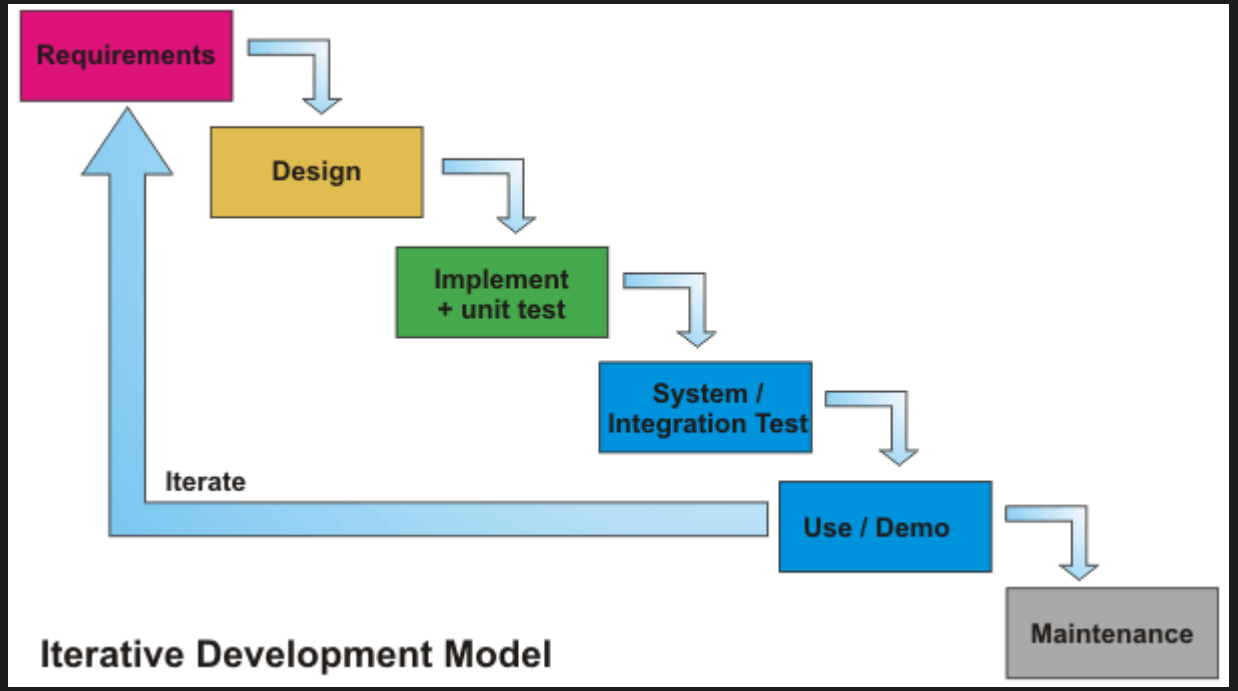
Calculating SDLC Risks

Day 182: “Oh crap. It turns out the fundamental library at the bottom of our stack doesn’t work.”

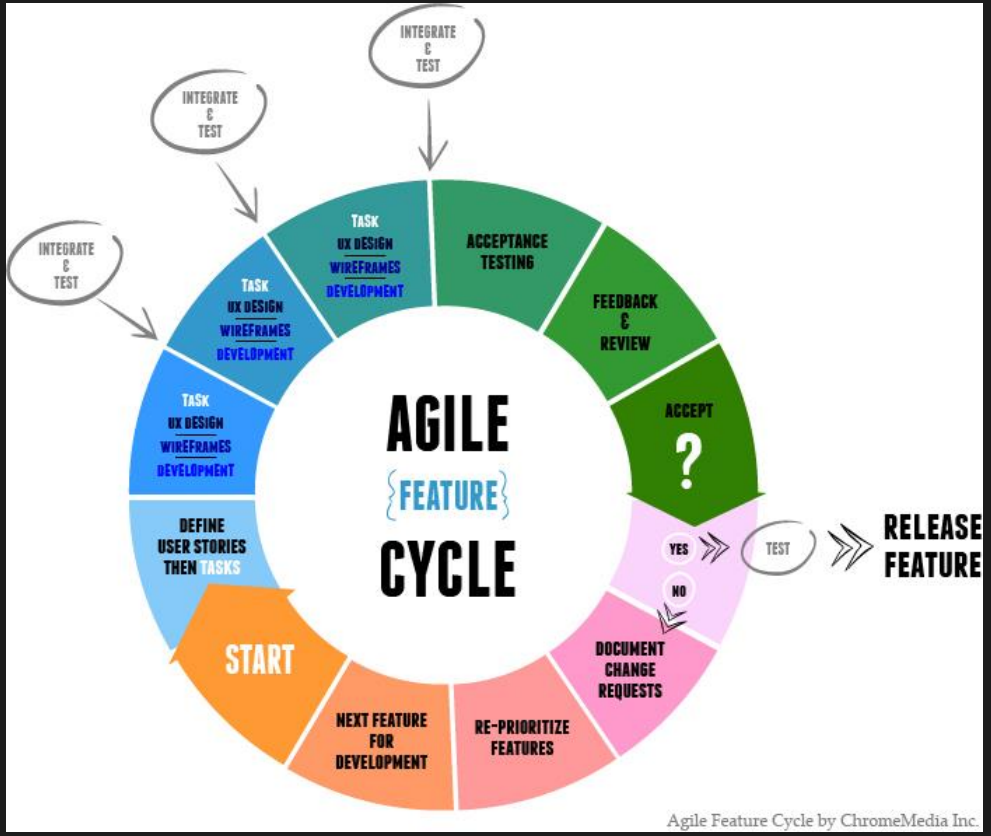
Spiral Development model



Iterative Development model



Agile Development model



Dealing with Risk

Initiate risk Identification sessions

- Are they real?
- Are they important?
- What action should we take?

Four options

- Avoid it – alter project requirements
- Confine it – make sure it affects a small part of the system or team
- Mitigate it – do something to see how the risk materializes, then eliminate it.
- Monitor it – come up with a contingency plan, deploy solution if risk gets out of acceptable bounds

Working with Risk

What are your risks

- Look at use cases and misuse cases (will talk about this in a bit)
- What can go wrong with implementing the use cases or what can bad actors do?
- How might the problem affect stakeholders
- Assess your team's capabilities

Prioritize risks and perform activities to assess/reduce risk

- Evaluate new technologies before use
- Rank risks by (perceived) exposure level. Address high impact/probability risks first

Part 2: Software requirements, use cases, and misuse cases

So you have this great idea…

Lets build a tower!

But there are Design / Implementation Issues then you get →

What are Software Requirements?

Criteria which the system or business must adhere to

- Thousands of “shoulds,” “shalls” and “musts” that detail how a system or piece of software operates
- Often form the contract between a customer and developer(s)
  - Necessitates formality and rigidity
- Created before coding begins, even in iterative and incremental processes
- Can be functional or non-functional

Example functional requirements

The facebook feed shall present feedlings to the user.

The facebook post UI must provide the ability to upload images.

The facebook post UI must provide the ability to tag other users.

Example non-functional requirements

The web service will be available for, at a minimum, 99% of the contract period.

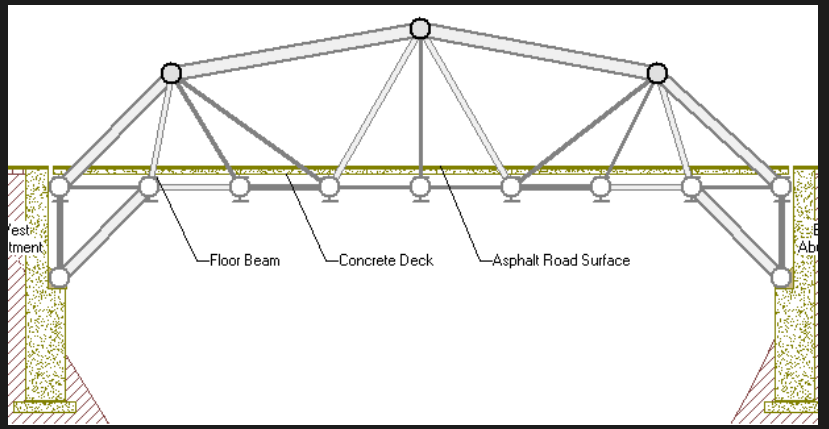
Software Requirements: Physical example (a bridge)

Functional

- The bridge must allow cars to pass over the stream
- The bridge shall have a separate area for pedestrian crossing
- The bridge shall allow small boats to pass under.

Nonfunctional

- The bridge will not buckle under a weight of 15000 pds or less
- The bridge will not sway or flex in winds under 75mph



Characteristics of good Software Requirements

Complete

Comprehensive and not open-ended

Testable

Can create a test for all requirements to determine conformance

Consistent

Should not be in conflict with each other

Design Free

Should be specified from a business perspective or need rather than the software perspective or what can be done

E.g.

The facebook post UI shall provide the ability to tag other users using the @ symbol. (good)

The facebook post UI shall autocomplete usernames typed after the @ symbol. (good)

vs

The facebook post UI shall interconnect with the postgres user database and utilize jQueryUI Autocomplete for @-based tagging of other users. (bad)

Unambiguous

Use “should,” "shall,“ or “must” followed by an active verb for a functional requirement or a performance expectation for a non-functional requirement

Software requirements: security versions (will return)

Security Functional Requirements (SFRs)

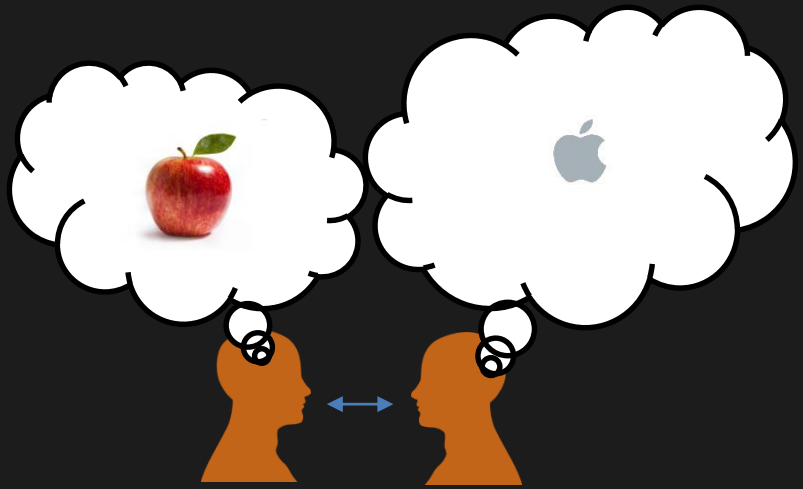
All software requirements that require some specific security functionality
e.g. The app shall *encrypt all messages* passed between the client and server.

Security Non functional Requirements

All non functional requirements that create C.I.A. security constraints.
e.g. The app shall be accessible 99% of the time.

Communication is 

Requirements are written by developers. But they rely on customer/management communication to gather and understand business/org. needs. Misinterpretation is easy.



User Stories

Narrative text of an interaction of the user with the system

- Focuses on the value a user gains from the system rather than the explicit interaction details
- Expanded on at the time of implementation for just-in time definition
- May include multiple use cases and/or require multiple functional requirements to be realized

User Stories: Desirable attributes

Independent

Reduced dependencies = easier to plan

Negotiable

Details added via collaboration

Valuable

Provides value to the customer

Estimable

Too big or too vague = not estimable

Small

Can be done in a reasonable amount of time (1-2weeks) by the team

Testable

Good acceptance criteria

User Stories: Format

"As a [*user role*], I want to [*goal*] so I can [*rationale*]"

It should fit on a 3"x5" card (in marker).

If it's too long

- It might be multiple stories
- You might be including too much detail more suited for the use cases instead of user story

Acceptance Criteria

Describes "how will I know when I'm done with the story"

Customer helps to determine criteria

May change as the functionality as the story matures

User Stories: Example 1

Title: Tag other users in facebook posts

Description:

As a facebook user, I need the ability to tag other facebook users in posts, so that my posts end up on their wall.

Acceptance criteria:

- A list of taggable users pops up when I type the @ symbol
- When I select a user from the list a tag to that user shows up in the post
- When I post a post with a user tagged in it, the post is copied to that user's wall.

Compare to the example use case (next)

this is much more focused on the value to the user instead of the specific sequence of interactions

User Stories: Example 2

Title: Review Performance of Advertising

Description:

As a vice president of marketing, I want to select a holiday season to be used when reviewing the performance of past advertising campaigns so that I can identify profitable ones

Acceptance Criteria:

- It should work with major retail holidays: Christmas, Easter, President's Day, Mother's Day, Father's Day, Labor Day, New Year's Day.
- Holiday seasons can be set from one holiday to the next (such as Thanksgiving to Christmas).
- Holiday seasons can be set to be a number of days prior to the holiday.

User Stories: Common Mistakes

Too much information!

- Loss of collaboration, loss of placeholder status
- No longer talking points or abstract to-do-list, but closer to requirements and/or use cases

Missing information within acceptance criteria

User satisfaction not fully understood

Use cases: A tool for describing interaction

Detail the interactions of the user (Actor) with the system in a call / response format

Format

The *use case begins* when the *user performs X*
The *system responds* by performing *X'*
…
The *user performs I*
The *system responds* by performing *I'*
…
The *user performs N*
The *use case ends* when the *system responds* by performing *N'*

Can be easily diagramed

Use cases: Example

Facebook post tagging

The *use case begins* when the *user clicks* “whats on your mind”.
The *system responds* by displaying the post UI box.
The *user types* “@<string>” in the post box.
The *system responds* by displaying a list of autocomplete possibilities that match the string
[Alt 1] The *user selects* a user from the autocomplete possibilities.
The *use case ends* when the *system responds* by inserting the selected user into the post.
[Alt 2] The *user continues to type*
The *use case ends* when the *systems responds* by hiding the list of autocomplete options.

Comparison between requirement statements

Basic difference is perspective and intent that affect the level of detail

Requirements

- Limit interpretation
- Level of detail can vary
- Requirements engineering processes are different
- Focus on system, not on user
- Successively refined

Use Cases

- User to system interaction perspective
- Capabilities are described
- Workflows can result across multiple use cases in a diagram
- Test cases can be written directly from use cases
- Successively refined

User Stories

- Focus on customer value
- Encourage specification generalities of the work to be done
- Collaboration pushes the story to implementation functionality
- Level of detail is ultimately more broad than a use case but not as rigid as a requirement

Lets look at some example user stories and use cases

Register

As an anonymous user, I want to be able to register, so that I can become a subscriber.

Use forum

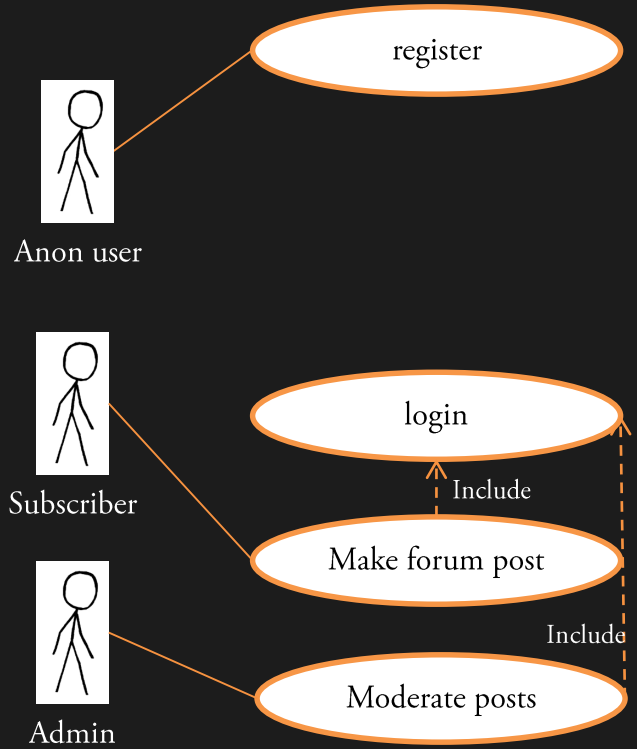
As a subscriber, I want to be make forum posts, so that I can collaborate with my team.

Moderate posts

As an admin, I want to be moderate the forum, so that subscribers use the forum appropriately and don't become unruly.

Lets look at an example use case diagram (a forum app)

Enumerating use cases is just thinking about all of the different scenarios in which an actor (user) uses your app.

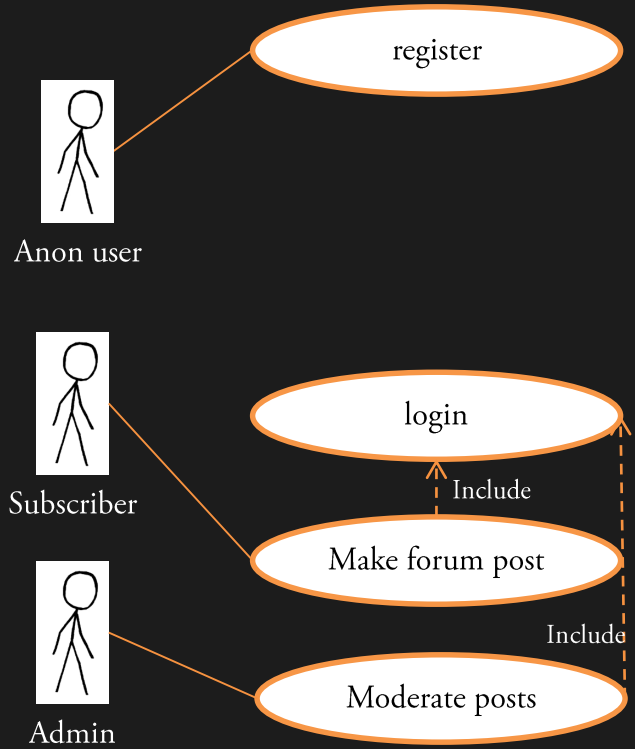


Register

The *use case begins* when the *user clicks* "register".
The *system responds* by displaying a form UI.
The *user enters* a username, password and email.
[Alt 1] The *system responds* by creating an account
[Alt 2] The *systems responds* by showing an error message.

Lets look at an example use case diagram (a forum app)

Enumerating use cases is just thinking about all of the different scenarios in which an actor (user) uses your app.

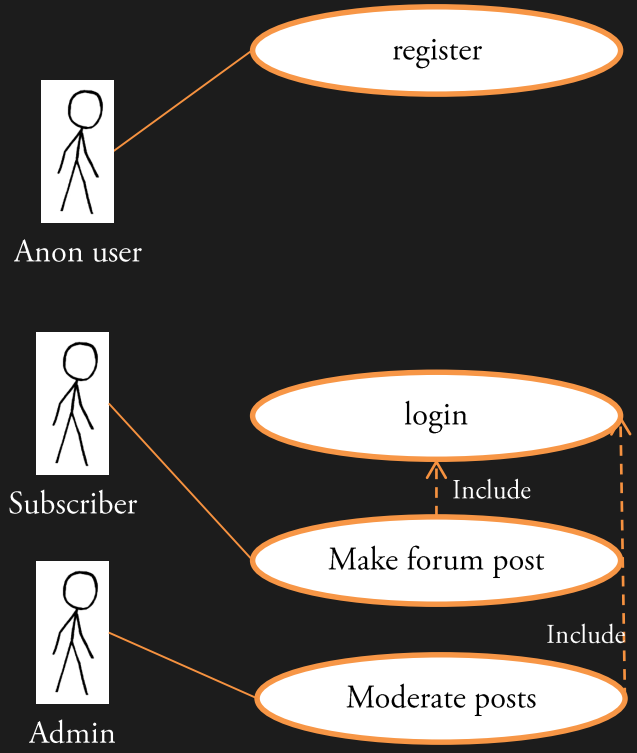


Login

The *use case begins* when the *user clicks* “login”.
The *system responds* by displaying a form UI.
The *user enters* a username and password.
[Alt 1] The *system responds* by logging the user in
[Alt 2] The *systems responds* by showing an error message.

Lets look at an example use case diagram (a forum app)

Enumerating use cases is just thinking about all of the different scenarios in which an actor (user) uses your app.

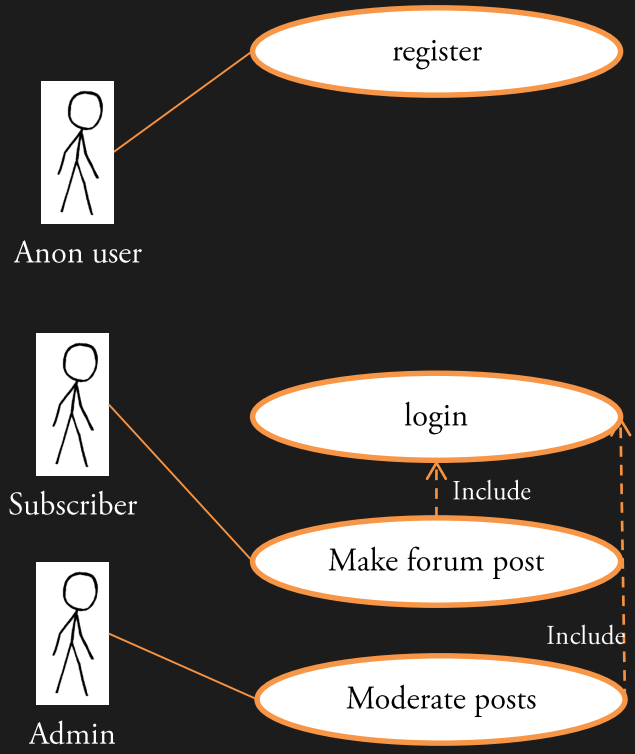


Make forum post

The *use case begins* when the *user clicks* “new post”.
The *system responds* by displaying a form UI.
The *user enters* a title and message.
[Alt 1] The *system responds* by posting the message.
[Alt 2] The *systems responds* by showing an error message.

Lets look at an example use case diagram (a forum app)

Enumerating use cases is just thinking about all of the different scenarios in which an actor (user) uses your app.



Moderate posts

The *use case begins* when the *user clicks* “moderate”.
The *system responds* by displaying an existing post.
The *user enters* makes changes to the post.
The *system responds* by saving the modified post and adding the string “edited by <username> @ <time> at the end of the post.

What about Security?

Pragmatism Pays

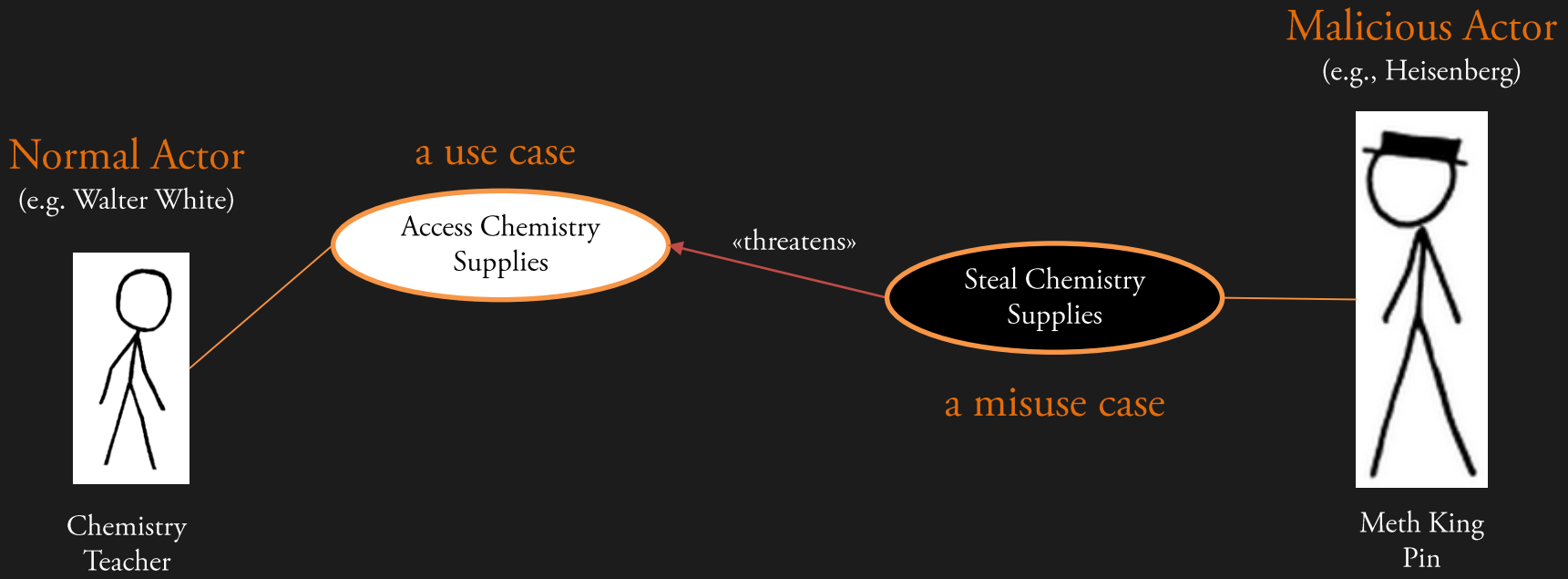
You need a balanced approach to thinking negativity.

think about what can go wrong and adapt to it

but don't lose sight of the project goals

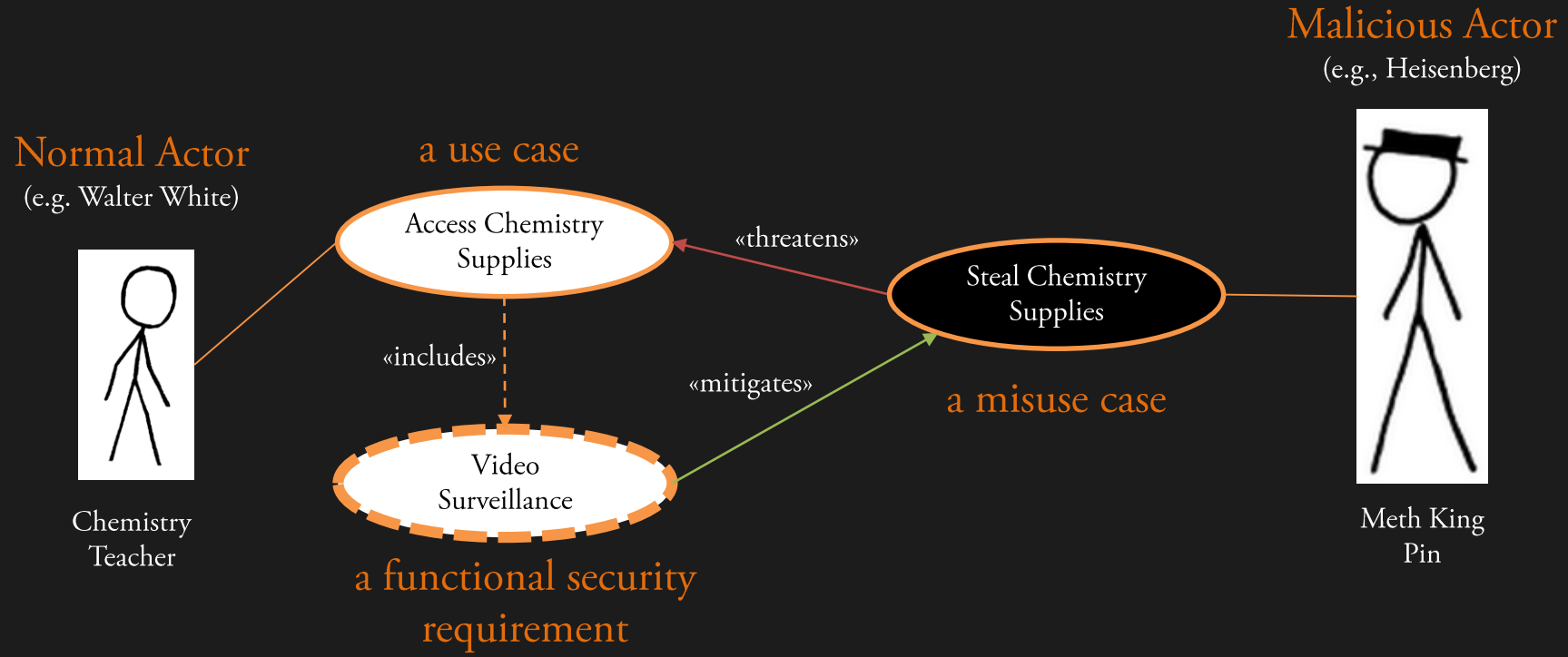
Enter misuse cases

Misuse cases imagine bad actors interacting with your system/software/app to abuse your actual use cases. They are expressed with «threaten» *stereotypes* (a UML concept) and inverted boxes.



What's the point?

The goal of eliciting and enumerating misuse cases is to determine a set of functional security requirements to mitigate the misuse cases.



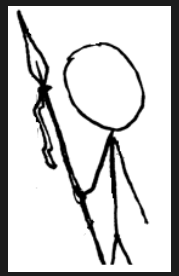
Misuse cases are basically just negative scenarios

A *scenario* is a series of actions that leads to a desired goal for an actor

A *negative scenario* is a scenario whose goal is either:
not desired by the organization controlling the scenario
or desired by a hostile actor (not necessarily human)

Thinking about what could go wrong and acting on it is not new…

Say Edward, I want to use this cave, but Bears are bad.

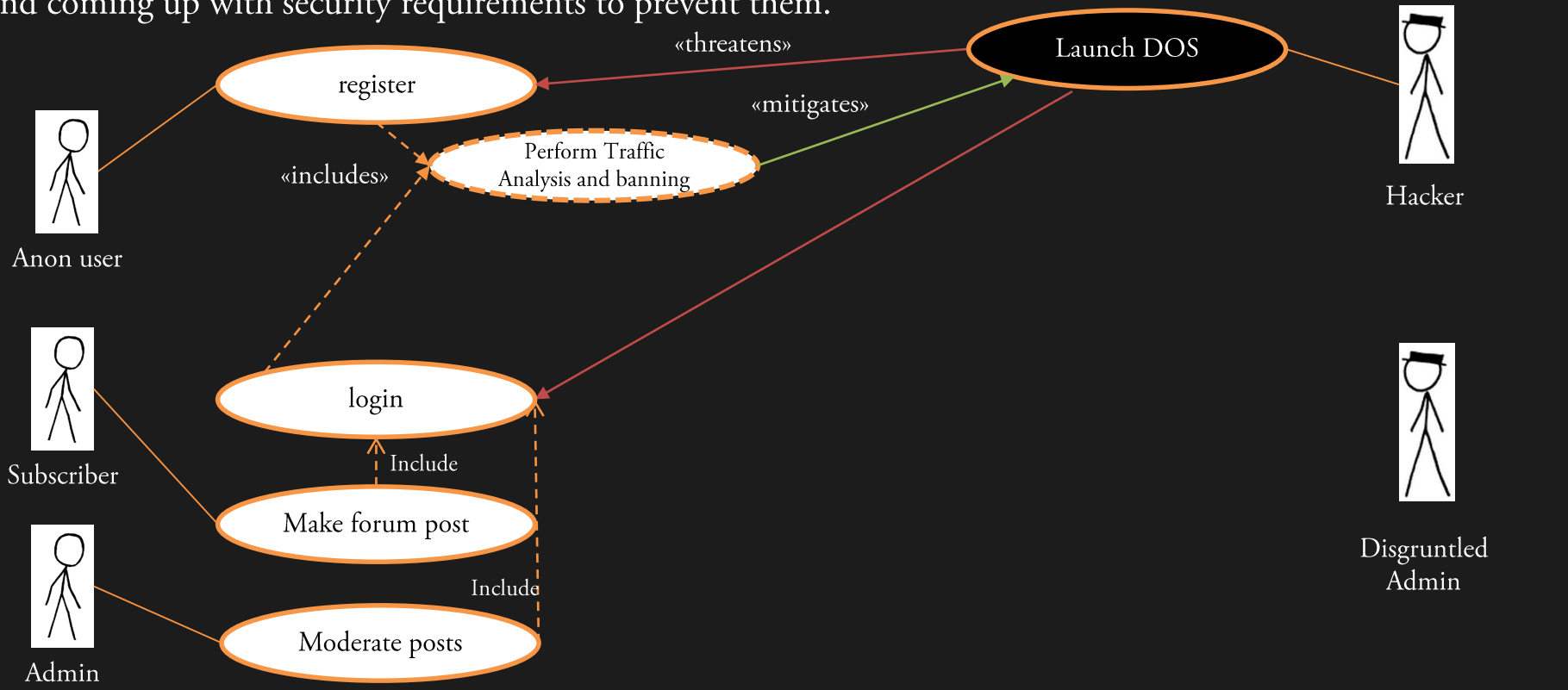


Brilliant! Right you are Phillip! Lets build a tent.

Cave painting of bears in Chauvet France circa 30,000 BC

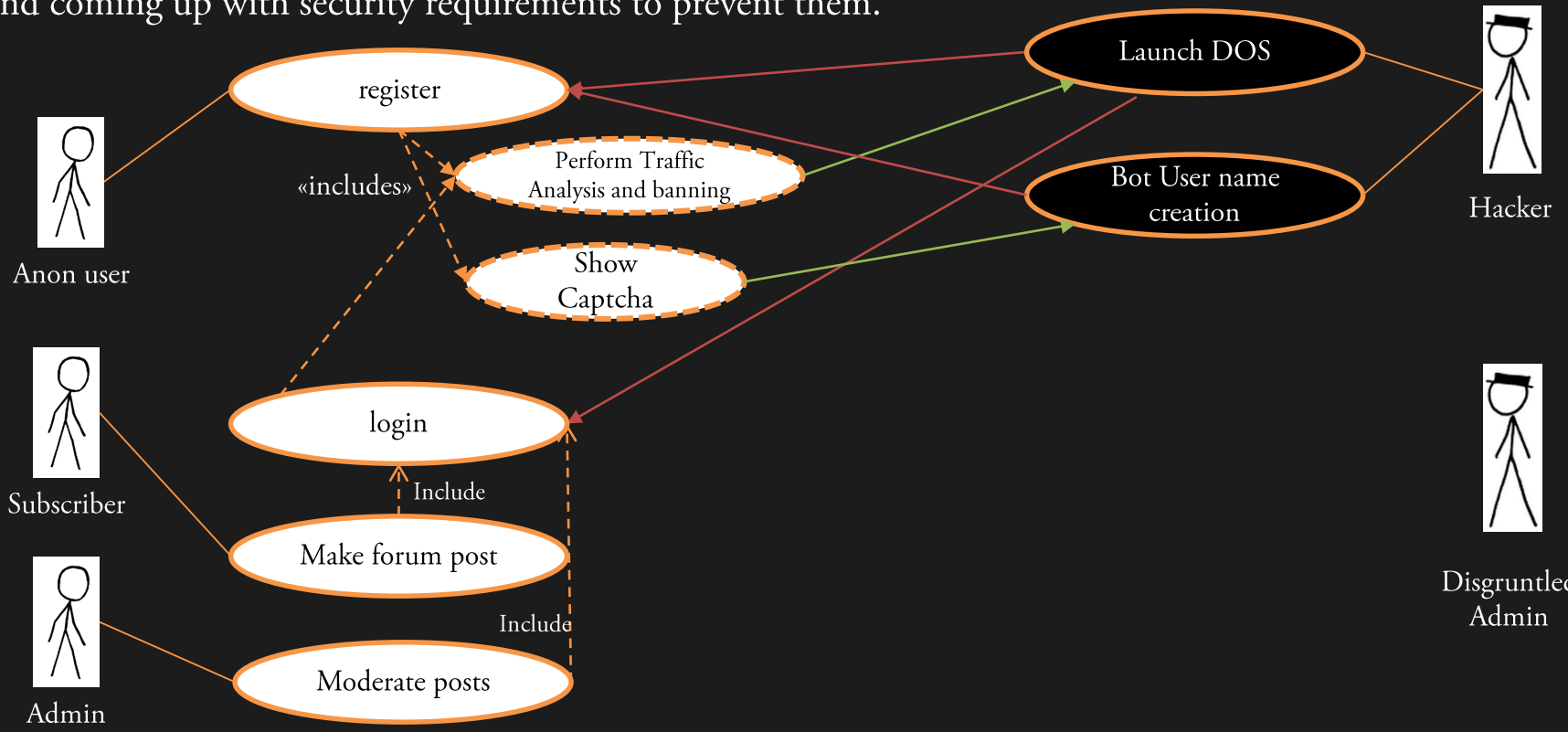
Lets look at a system example (a forum)

Enumerating misuse cases is just thinking about all the negative scenarios that could go wrong with a use case and coming up with security requirements to prevent them.



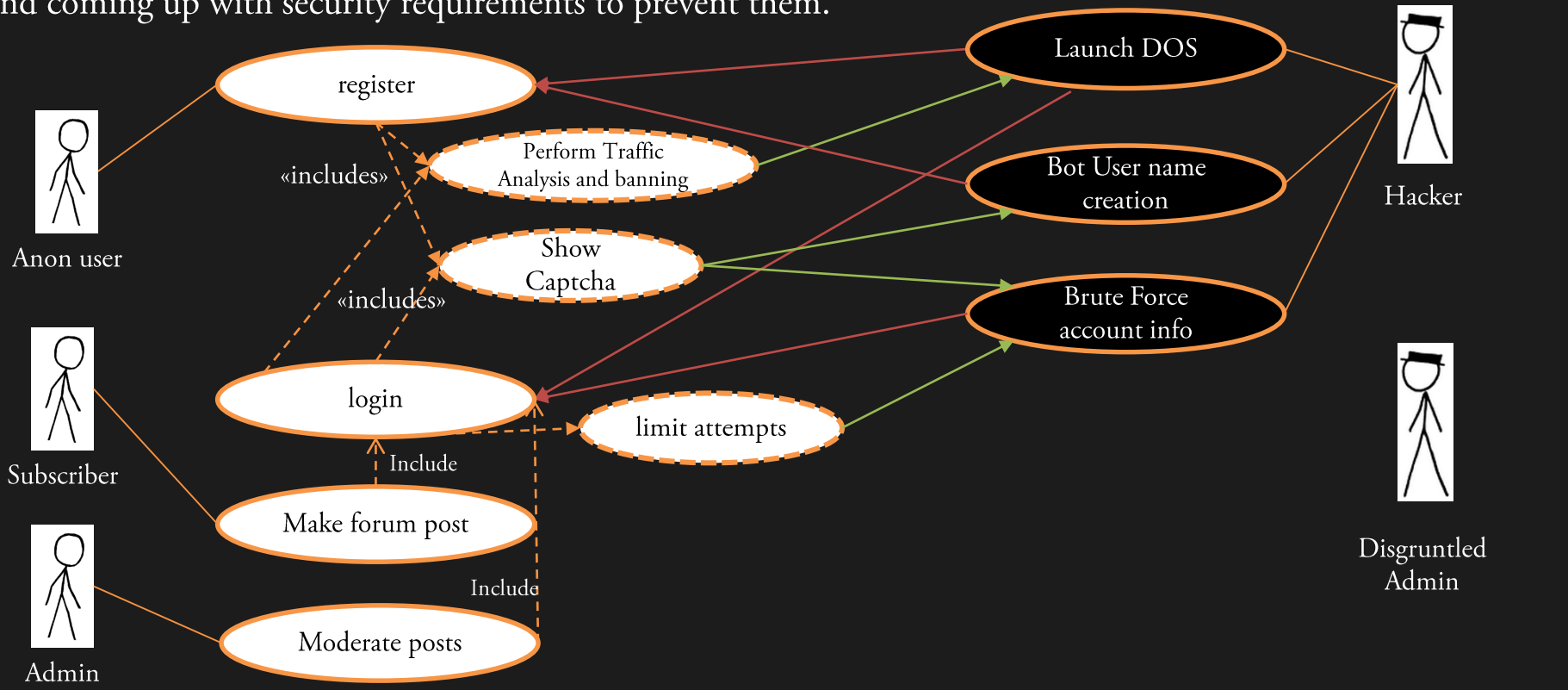
Lets look at a system example (a forum)

Enumerating misuse cases is just thinking about all the negative scenarios that could go wrong with a use case and coming up with security requirements to prevent them.



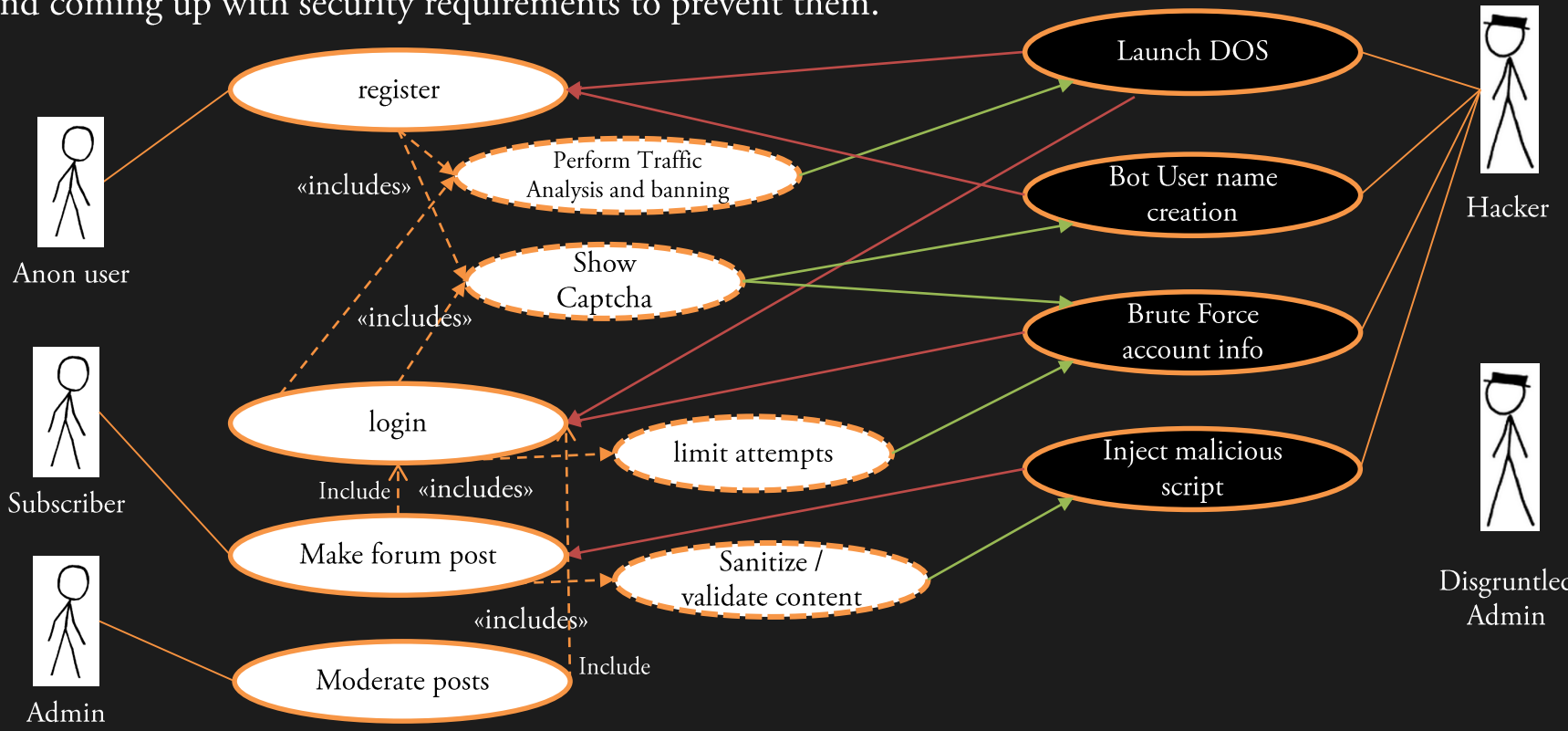
Lets look at a system example (a forum)

Enumerating misuse cases is just thinking about all the negative scenarios that could go wrong with a use case and coming up with security requirements to prevent them.



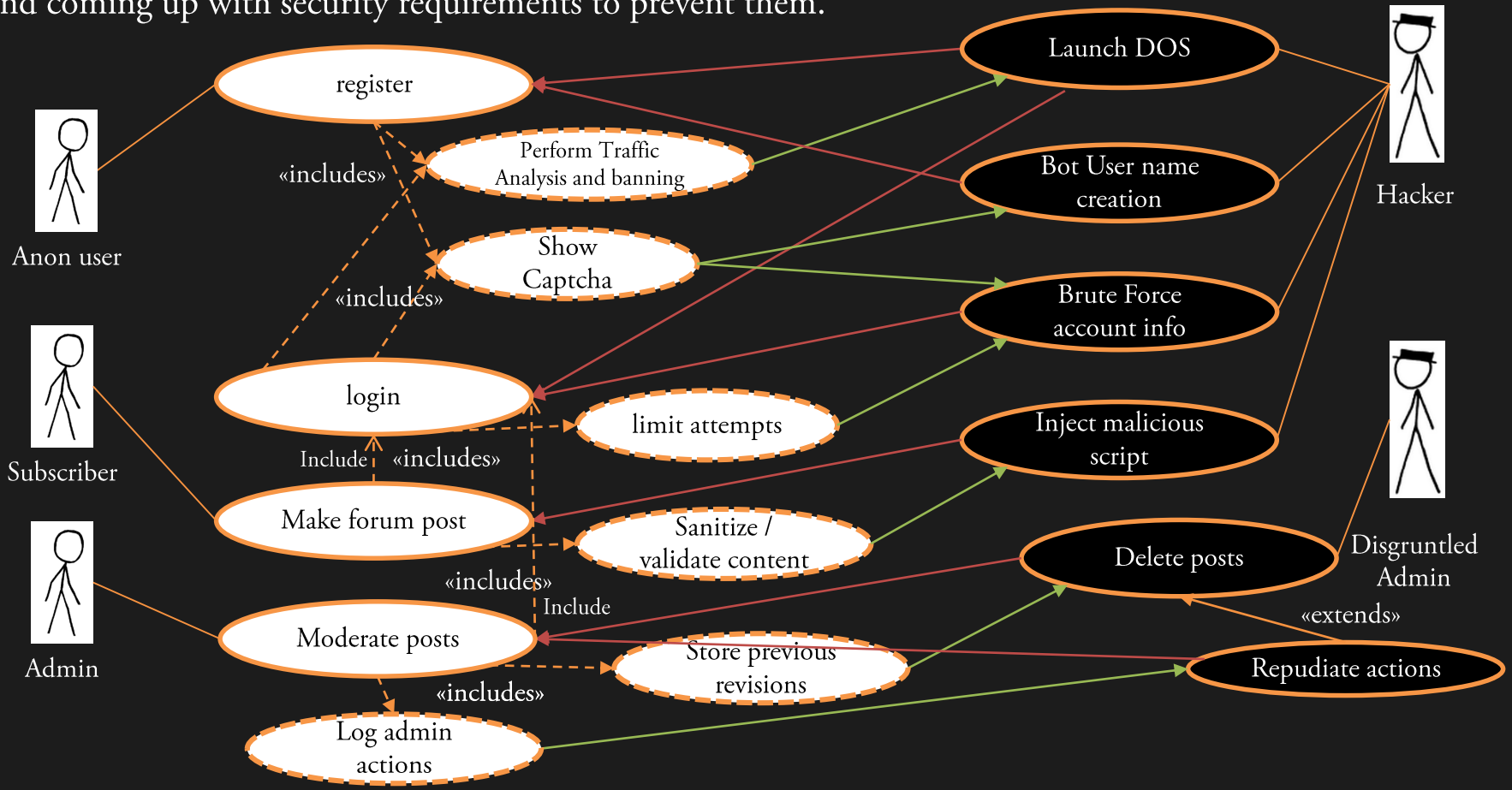
Lets look at a system example (a forum)

Enumerating misuse cases is just thinking about all the negative scenarios that could go wrong with a use case and coming up with security requirements to prevent them.



Lets look at a system example (a forum)

Enumerating misuse cases is just thinking about all the negative scenarios that could go wrong with a use case and coming up with security requirements to prevent them.



Great thought tool – helps identify potential problems

As a developer or security consultant always try to exhaustively define misuse cases.

Bottom Line: Focus on the dotted ellipses

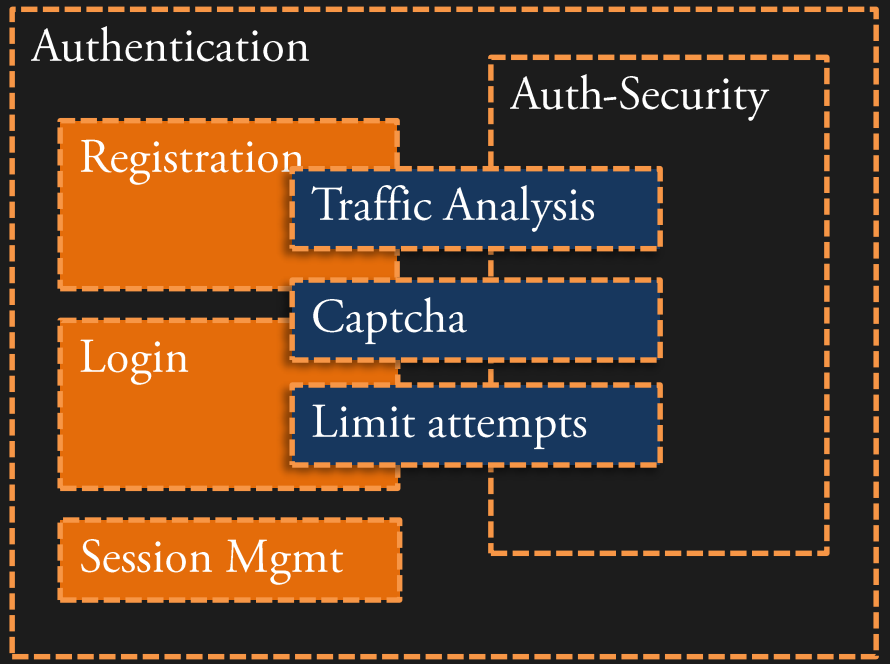
By countering misuse cases you create a list of functional security requirements for your system. Implement these (correctly) and you mitigate the misuse case.

This, unfortunately, doesn't mean the component is secure – you don't know what you don't know.

→ Misuse cases *almost always* create functional security requirements

Functional security requirements are realized by building components

Security centric components should be built and integrated into application architectures. Generally it's a good idea to re-use existing frameworks and integrate instead of reinventing core security functionality like action logging, traffic monitoring, DDoS protection, etc.



Some security requirements (and components) are re-usable

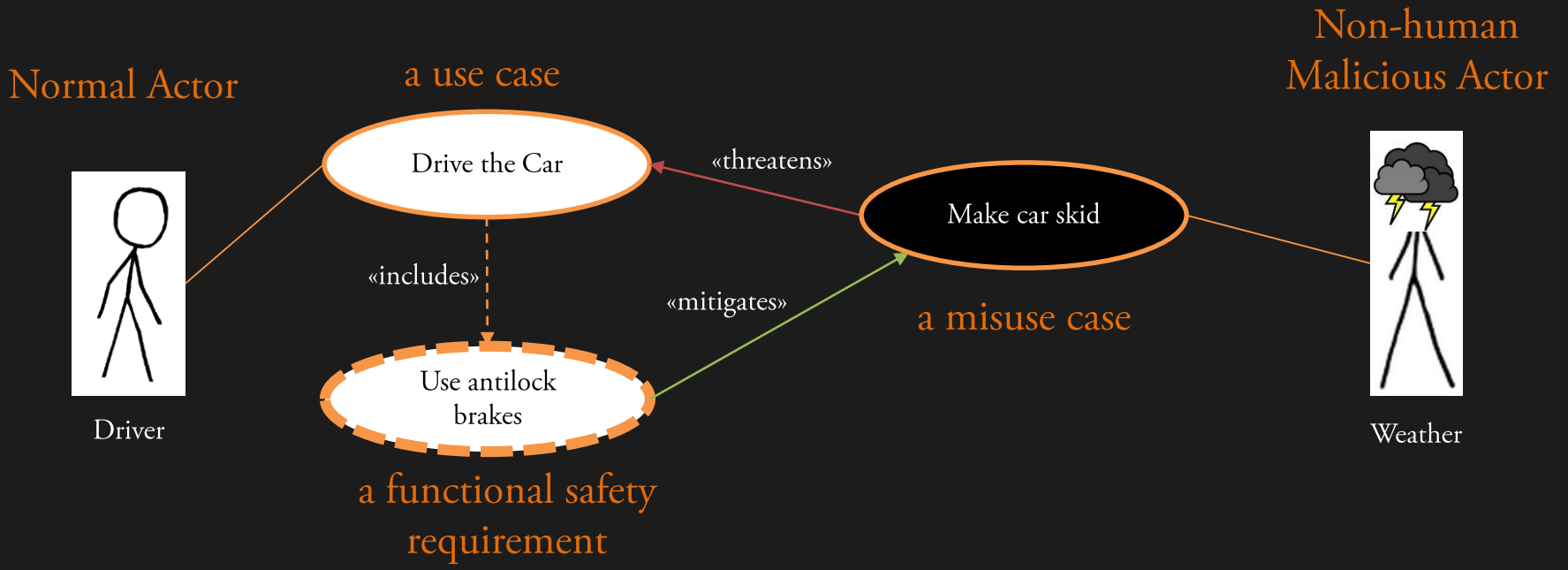
There are tons of regulatory and industry standards that specify misuse cases and functional security requirements. The CVE (Common Vulnerability Enumeration) and CWE* (Common Weakness Enumeration) as well as the NVD (National Vulnerability Database) are all good resources for helping you to think about different misuse cases depending on what software/hardware you are using.

Documents like ISO 27002, Common Criteria (Part 2), and NIST SP800-53* all provide *security controls* which specify functional (and non-functional) security requirements for certification purposes.

*You may have heard about these in Dr. Gandhi's classes or my policy class

Other Applications – Misuse cases for Safety Requirement Elicitation

Anthropomorphize actors that aren't human (e.g., the weather). What bad actors exist? Answer the question to elicit safety requirements.



Questions?

Matt Hale, PhD

University of Nebraska at Omaha

Interdisciplinary Informatics

mlhale@unomaha.edu

Twitter: @mlhale_